Grammar Transformations for Comparing Modelica Versions

July 1, 2013

Tanya Braun

supervised by:
Prof. Dr. Sibylle Schupp
Dipl.-Ing. Timo Kamph

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Eidesstattliche Erklärung

Ich versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 01.07.2013

______________________________
Tanya Braun
E.13 Refactoring of Access Expressions ......................... 114
E.14 Refactoring of Named Arguments .............................. 115
E.15 Refactoring of Partial Function Calls ......................... 115
E.16 Refactoring of Function Calls and Function Arguments ............ 116
E.17 Elimination of Redundant Choices .............................. 116
E.18 Final Refactorings for Clarification of Differences .............. 116
E.19 Renaming of Terminals ......................................... 117
E.20 Refactoring of Subscripts ...................................... 118
E.21 Refactoring of Expression Lists ............................... 118
E.22 Refactoring of Relational Operators .............................. 119
E.23 Refactoring of Multiplicative Operators ......................... 119
E.24 Refactoring of Unary Expressions .............................. 119
E.25 Refactoring of If Expressions ................................ 120
E.26 Refactoring of Output Expressions ............................. 120
E.27 Refactoring of Matrices ....................................... 121
E.28 Refactoring of References ..................................... 121
E.29 Refactoring of For-or-expression Lists ......................... 122
E.30 Refactoring of Function Arguments ............................ 122
E.31 Refactoring of Function Call Arguments ....................... 123
E.32 Refactoring of Function Expressions ......................... 124
E.33 Refactoring of Simple Expressions ............................ 124
E.34 Elimination of Redundant Choices ............................. 124
E.35 Elimination of Names in OpenModelica ......................... 125
E.36 Elimination of Names in JModelica ............................ 125
E.37 Refactoring of JModelica ..................................... 125
### Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Final Converging and Failing Productions between the Modelica specification and JModelica</td>
<td>33</td>
</tr>
<tr>
<td>4.2</td>
<td>Final Converging and Failing Productions between the Modelica specification and OpenModelica</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Final Converging and Failing Productions between OpenModelica and JModelica</td>
<td>44</td>
</tr>
<tr>
<td>5.1</td>
<td>Summary of the Differences between the Expressions in the Modelica Specification and JModelica</td>
<td>48</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary of the Differences between the Expressions in the Modelica Specification and OpenModelica</td>
<td>48</td>
</tr>
<tr>
<td>5.3</td>
<td>Summary of the Differences between the Expressions in OpenModelica and JModelica</td>
<td>48</td>
</tr>
<tr>
<td>5.4</td>
<td>Overview of the Transformations Used in the Three LCIS</td>
<td>63</td>
</tr>
</tbody>
</table>
## Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Parse Tree of the Modelica Specification Expressions</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Parse Tree of the JModelica Expressions</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>Parse Tree of the OpenModelica Expressions</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Abstract Convergence Process</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Convergence Process of the Example LCI</td>
<td>21</td>
</tr>
<tr>
<td>5.1</td>
<td>Parse Tree of the Transformed JModelica Expressions</td>
<td>50</td>
</tr>
<tr>
<td>5.2</td>
<td>Parse Tree of the Transformed OpenModelica Expressions</td>
<td>51</td>
</tr>
<tr>
<td>5.3</td>
<td>Parse Tree of the Modelica Specification Expressions</td>
<td>52</td>
</tr>
</tbody>
</table>
Nomenclature

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGF</td>
<td>BNF-like Grammar Format</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus Naur Form</td>
</tr>
<tr>
<td>DNF</td>
<td>Disjunctive Normal Form</td>
</tr>
<tr>
<td>EBNF</td>
<td>Extended Backus Naur Form</td>
</tr>
<tr>
<td>LCF</td>
<td>LCI Configuration Format</td>
</tr>
<tr>
<td>LCI</td>
<td>Language Convergence Infrastructure</td>
</tr>
<tr>
<td>SLPS</td>
<td>Software Language Processing Suite</td>
</tr>
<tr>
<td>XBGF</td>
<td>Grammar Transformation Language; transformations (X) of grammars in BGF</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Modelica, an object-oriented programming language for modeling physical systems, is the outcome of the PhD thesis by Hilding Elmqvist published in 1978 [1]. Almost twenty years later, the ideas presented in the PhD thesis were picked up in an effort to design a language for modeling physical systems. In September 1997, Modelica 1 was published [2]. Three years later, the non-profit organization Modelica Association was founded to manage the development of the Modelica language [3]. Commercial and open-source front-ends started to evolve. JModelica is an open-source platform that handles optimizing, simulating, and analyzing dynamic systems. OpenModelica is a simulation and modeling environment for Modelica.

The Modelica language specification includes a syntactic description of the language. The front-ends usually work on a grammar ingrained in the respective parser descriptions. With the different descriptions the Modelica language receives, the question arises whether these descriptions model the same language. Working in an area of computer science, you may come across a variety of software artifacts that include grammar knowledge. For Modelica, this grammar knowledge is sourced in parser descriptions and a language specification document. For reasons like interoperability, this knowledge should be consistent. One problem is to establish and maintain the relationship between the grammar knowledge in the different software artifacts.

The problem we tackle in this thesis is the correspondence between the different Modelica grammar versions. To find an answer to this problem, we compare the grammars. With the help of a suitable tool, we want to establish a relation between the grammars. As the versions, by their own nature, vary, we will need to transform the grammars to align them. Grammar transformations is our means to narrow down the differences between the grammar versions.

The result of this cycle of transformations and comparisons gives us an answer to the question whether the grammars model the same language. If the process discloses any differences that cannot be resolved, the question then becomes whether the difference is evidence of a contradiction between the two grammars or the case of one grammar being more relaxed than the other. In addition, one might ask what the origins of the difference could have been. In this thesis, we want to answer these questions for an excerpt of three grammar versions of the Modelica language. The grammar versions come from the Modelica language specification and the Modelica implementations JModelica and OpenModelica.

This thesis follows a road map towards answering the question whether the different Modelica grammar versions model the same language. The road map starts at the foundations of the thesis in Chapter 2. First, we take a closer look at the different Modelica versions in Section 2.1. We introduce the language Modelica and its simulation environments JModelica and OpenModelica.
and examine how each version presents its grammar. Additionally, we reflect on the grammars’ parse trees and carry out a superficial comparison of the trees. Second, we draw a picture of the current state of the art in comparing grammars and converging languages in Section 2.2. This overview considers the background and motivation of language convergence, the current state, and related work. With the current state, we meet the means to answer our question, the tool Language Convergence Infrastructure. The overview also takes a quick glance at open research questions. From the foundations, we move on to the tool we have chosen to investigate the Modelica grammar versions with in Chapter 3. Its setup, its components, and a simple example are the topics visited within this chapter. Next stop on the road map is the pairwise comparison of the grammar versions using the Language Convergence Infrastructure in Chapter 4. We discuss the grammar transformations applied as well as the results of the comparisons. An evaluation of the outcome of the investigation follows in the subsequent chapter, Chapter 5. It includes an analysis of the results (Section 5.1), a discussion of the transformations used (Section 5.2), and considerations about the Language Convergence Infrastructure as the means to answer our question (Section 5.3). The road map ends at Chapter 6 with the conclusion extended by some thoughts about future work.
Chapter 2

Foundations

In this chapter, we familiarize ourselves with background information concerning Modelica and its grammar versions as well as with the current state of the art relating language convergence and grammar comparison. We start with the Modelica versions in Section 2.1 and go on to the overview of language convergence afterwards in Section 2.2.

2.1 Modelica Versions

Modelica is an object-oriented language for modeling systems, e.g., mechatronic models in robotics or aerospace applications with a large number of subsystems. All parts of a model can be devised in the language. Modelica has a mathematical nature: all components of the model description are mapped to differential, algebraic, and discrete equations.

To fully utilize Modelica, a simulation environment is required. An environment supports the user in defining Modelica models in graphical user interfaces, which translate a graphical representation into a textual Modelica model. The environment translates the model into a form it can simulate. Finally, it simulates the model and visualizes the results. Of the environments available, two noncommercial, open-source projects are JModelica.org and OpenModelica.

The information on Modelica, JModelica, and OpenModelica originate from the corresponding user’s guides and language specifications as well as API documentations. For Modelica, the Language Specification document [4] is the source of information. The JModelica knowledge comes from the user’s guide [5] and the API documentation [6]. The insight in OpenModelica stems from the user’s guide [7].

The remainder of this chapter consists of three sections, each concentrating on one Modelica grammar version, starting with the Modelica specification itself.

2.1.1 Modelica Specification

Modelica states the language’s grammar in its Language Specification document. The most recent version is 3.3 but the grammar has not been changed from version 3.2 to its successor. The grammar is expressed in extended Backus Naur Form (EBNF) that is optimized for readability. Syntactic meta symbols exist to indicate a disjunction, a quantity of zero or one, and an iteration of zero or more times. Square brackets denote an optional expression. Curly braces mark an expression to be repeated zero or more times. A vertical bar separates choices from each other. Text in quotation marks is treated as a single token. An example production using these meta
CHAPTER 2. FOUNDATIONS

symbols is the following choice of either an optional terminal named “text” or a star expression of a nonterminal named “body”.

head :
    [ "text" ] | { body }

An excerpt of the grammar, namely, the section “Expressions”, can be found in Appendix A.1. A slight difference exists between the grammar in the language document and in the appendix. In the published version, Modelica keywords are in boldface, a markup lost after copying text from a PDF file. The keywords are reserved and, therefore, may not be used as identifiers. But as part of the grammar, they are terminals and by their nature, treated as a single token. As a consequence, the keywords are not in boldface but in quotation marks in the appendix.

Let us have a quick look at the structure of the grammar fragment in the Appendix. Figure 2.1 depicts the corresponding parse tree. We can see from the figure that the tree consists of two parts, one starting and ending at the node `expression` and one for comments. From the node `expression`, a relatively direct branch leads to the node `primary` after which the tree splits up into several branches of which most lead back to the node `expression`.

After the grammar from the Modelica specification has been introduced, we will now take a look at the grammars of the two Modelica implementations, JModelica and OpenModelica.

2.1.2 JModelica

JModelica.org provides a simulation environment for Modelica. Its most recent release claims to support Modelica 3.2. Its grammar is specified through a parser description, which can be found in the JModelica repository1. The parser is generated by Beaver, a parser generator for deterministic context-free languages. Beaver generates LR parsers that parse the input from left to right and construct a rightmost derivation. The input grammar needs to be in EBNF [8]. The generated parser consists of parser-specific code, including error handling procedures. The production heads have a class, the referenced nonterminals can be identified by variable names, and each branch has a return statement. The JModelica parser uses only two syntactic meta symbols for expressing the grammar. Vertical bars separate choices, a question mark denotes an optional expression. Terminals are in uppercase letters. The same example production as in Modelica reads using JModelica meta symbols:

head :
    TEXT? | body_star?
body_star :
    body body_star

JModelica does not use a star expression, therefore, two productions are needed to model this kind of expression. A version of the grammar used in the parser is presented in Appendix A.2. It contains only the productions that model the same expressions as the Modelica expressions and is converted to EBNF for readability.

The parse tree for the expressions in the JModelica grammar is pictured in Figure 2.2. Again, we have a split tree for expressions and comments. The whole tree appears longer, more nonterminals belong to the tree. From the node `exp`, a direct branch leads to the node `primary`. Similar to the Modelica specification parse tree, most branches originating from the node `primary` return to the node `exp`. For example, the branch originating from the node `access_expression` ends in the same constructs, array subscripts, as the Modelica specification. The nodes in between model some kind of smaller access expression themselves. We may assume that this branch generates the component references from the Modelica specification.

1Parser description within the JModelica repository: https://svn.jmodelica.org/branches/1.9.x/Compiler/ModelicaFrontEnd/src/parser/Modelica.parser
Figure 2.1: Parse Tree of the Modelica Specification Expressions
Figure 2.2: Parse Tree of the JModelica Expressions
2.2. OVERVIEW OF LANGUAGE CONVERGENCE

2.1.3 OpenModelica

OpenModelica also implements the Modelica language and as such provides another simulation environment for Modelica. Its most recent release supports Modelica 3.2. Its parser description displays the grammar underlying the OpenModelica implementation. It is part of the OpenModelica source code. OpenModelica uses ANTLR to generate the parser. ANTLR is also a parser generator for context-free languages that takes grammars in EBNF as input. In contrast to Beaver, ANTLR generates LL parsers that read input from left to right and construct a left-most derivation [9]. The parser-specific code includes return statements, variable names, and instructions that update the abstract syntax tree. The meta symbols in the grammar notation are vertical bars, question marks, and stars, denoting a choice, an optional, and a star expression respectively. Uppercase letters mark the terminals. OpenModelica expresses the simple example production in the following way:

head : TEXT? | body

Appendix A.3 shows a selection of the parser description, i.e., the expressions, transformed into EBNF.

For the last grammar, we also survey the parse tree, shown in Figure 2.3. Similar to the previously considered parse trees, one tree starts with the node comment and the other tree has the node expression as the top node. Many of the branches stemming from the node expression route back to the node. The tree illustrates that the node expression calls more nodes than the corresponding node in the other two parse trees. A direct branch leads from the node expression to the node primary as in JModelica. The part after the node primary appears to be less complex compared to JModelica but uses more nodes compared to the Modelica specification. Another notable feature of the OpenModelica tree is that one branch originates from the node expression and bypasses a large portion of the branch that leads to the node primary. This branch is singular in the set of the presented parse trees.

With the short introduction into each grammar version, we have an idea of the overall concurrency and the immanent differences between the three versions. Next, we give a survey on the current state of language convergence.

2.2 Overview of Language Convergence

This section focuses on language convergence. More specifically, it concentrates on grammars and the convergence of grammars of the same language but from different sources or dialects. Modelica is a language with different sources for grammar knowledge which we want to converge.

Programming languages receive multiple descriptions. The grammars that are explicitly or implicitly embodied in these descriptions are ideally equivalent. But considering the vast nature of the sources, the fact that different compilers exist for the same language, or that languages evolve makes it arguable whether the grammars are indeed equivalent. The idea behind language convergence is to check whether these grammars are consistent. Examining consistency is beneficial for various reasons, among them avoiding problems when using different compilers or having correct language documentation. A main contribution to that research area has been made by Zaytsev and his co-authors [10, 11, 12, 13].

This section has the following structure: In the first subsection, we enlarge on the background and motivation for language convergence. The second subsection contains an overview of the
Figure 2.3: Parse Tree of the OpenModelica Expressions
current state of the work in the field and approaches taken to answer the question about consistency. Related work receives attention in the third subsection. An impression of future work and open research questions follows in the second to last subsection before we end this section with a summary and a quick outlook on our further procedure.

2.2.1 Background and Motivation for Language Convergence

As mentioned before, language knowledge deviates. What languages do and what they may not be able to do, how they look, and what logic they follow can be found in various sources. Language documentation, XML schemas, compiler and parser descriptions, or even actual source code comprise information about the language and how it works. An important ingredient of any programming language is its syntax. A grammar is a description of the syntax. Usually, the grammar is given in Backus Naur Form (BNF) or a variation of it. It is possible to extract grammars from different sources with varying complexity and success. As the grammars have different targets, they may differ from each other though they concern the same language. Furthermore, considering that different compilers exist for one language, grammars extracted from these sources do not naturally match. It is favorable to know about these differences to be able to handle them or to be able to rely on their absence.

Grammars may not only vary because they come from different sources but also on a lower level, from project work. People on the same project use different notations. At one point, the participants might want to ensure that their code is consistent. Within a broader scope, programming languages evolve over time as people around the world work with them, extend, or customize them. Compilers and parsers as well as language documentation have to keep up. A framework checking for differences between current versions, but also between current and previous versions, can help to identify changes.

The background for language convergence lays mainly in grammar transformations. The research in grammar transformation is vast and deserves an own overview. But to summarize briefly the main issues grammar engineers face, consider

- left recursion,
- ambiguity of grammars,
- the question whether transformations preserve the semantics of the grammar, and
- a high diversity in syntactic notation.

Left recursion can lead to nontermination of algorithms when applying a top down left to right approach, e.g., during parsing. Many researchers investigate finding efficient solutions to that problem, e.g., [14] and [15]. Basten [16] argues that ambiguities in grammars may be wanted but accidentally introduced ones can be harmful, and proposes a method to find them. Grammar transformations can be categorized by refactoring, construction, and destruction. Refactoring preserves the semantics of a grammar, like renaming a nonterminal or function. Construction and destruction change the behavior of a grammar. Lämmel has built a comprehensive operator suite for grammar transformations, which are tested and documented [17]. The suite presents a valuable resource to other peers as they can rely on tested transformations. The last issue in this list refers to a problem recognized as early as 1977 when Wirth [18] asked in an article: “What can we do about the unnecessary diversity of notation for syntactic definitions?” Although all notations are based on BNF, many different versions exist, e.g., WBNF in [19] and TBNF in [20]. A standardization effort by the ISO has only lead to other variants and not to a unification [21]. In an attempt to tackle the diversity from a different angle, Zaytsev proposed to keep the diversity and employ a meta language [22], named BGF, a BNF-like grammar format. The meta language should build a bridge between different frameworks and syntactic notations.

Work that has preceded the recent work on language convergence include
• parser generation [23],
• grammar recovery [24],
• grammar testing [25], and
• grammar transformation and adaption [26].

On a different level, researchers have spent time on the question of equivalence of abstract and concrete syntax. In contrast to language convergence where several sources are considered and checked, the equivalence regarding abstract and concrete syntax addresses various problems. In case a concrete syntax already exists, an abstract syntax must be constructed such that it matches [19]. If both syntaxes exist, a mapping needs to be established between the two syntaxes [27]. Researchers also advocate developing an abstract and a concrete syntax together to ensure consistency [23]. The current approaches do not start with abstract or concrete syntax but on a higher level.

2.2.2 Current State

To my knowledge, there are currently two main approaches to comparing extensive grammars. One involves a meta language to which the grammars are translated. The other one uses test sets generated from the grammars under investigation. We take a closer look at both of them in this section.

Language Convergence Using a Meta Language

Nurturing from previous work from his co-author Ralf Lämmel on grammar recovery [24] and adaption [26], Vadim Zaytsev published his work on language convergence [10, 11]. The idea behind his work is to verify the consistency of grammar knowledge encapsulated in all kinds of software artifacts. The tool chain consists of grammar extraction and automated grammar transformations of refactoring and editing until the grammars are syntactically identical. The grammars converge to broader and broader targets leading up to a concrete and an abstract syntax representation. The two syntax representations are ultimately converged to one limit, which itself is an abstract syntax. The grammar transformations form an own language, XBGF. From the transformations applied to the grammars, one can draw conclusions regarding contradictions, extension, inclusion, or equivalence. During extraction, the tool abstracts the grammar and converts it to BGF. The abstracted grammar format allows for simple grammar transformations.

Two years later, Zaytsev presented a language convergence infrastructure [12], a framework part of the open source project Software Language Processing Suite (SLPS) [28]. SLPS consists of scripts developed for comparison, transformation, benchmarking, validation, extraction, and pretty-printing of grammars. The framework combines a transformer and a comparator. For the process to work, the transformer and the comparator need the grammars in BGF. Extractors are scripts that process a software artifact and produce a BGF grammar. They are language-specific. For several languages, they already exist and are part of the suite. The transformer changes the grammars to make them structurally identical. It takes an XBGF script and a BGF grammar as input and generates a transformed BGF grammar as output by applying the input script to the input grammar. The comparator returns the result of the comparison of the grammar sources. Optional are a validator and a generator. The validator checks the XML files generated in the process of checking language convergence for being well-formed. A generator takes a BGF grammar as an input and produces an XBGF script applicable to that grammar. The script could be used as an input for the transformer.

As admitted by Zaytsev and Lämmel [10], using BGF implies losing some of the details that are part of other forms of grammars. The tool might miss some minor differences between
Grammars. Some of these details constitute an own research question, like precedence rules. For example, Bouwers et al. [29] developed a tool for comparing and recovering precedence rules and checking for compatibility. With the use of grammar transformations, their tool builds a common representation for the grammars.

**Grammar Comparison Using Test Sets**

The authors of the aforementioned approach also published another method in collaboration with Bernd Fischer [13]. In an automated approach, grammars are compared to find evidence supporting a claim for (non-)equivalence. The method compares context-free grammars based on parsing generated test data. It is also part of the project SLPS. The work uses results published by Lämmel on grammar testing [25]. It profits from previous work on grammar recovery [24] and language convergence [10] for grammar recovery.

The work includes a framework for matching nonterminals. The idea is to test acceptance/rejection of the generated test data and thereby, match nonterminals of two grammars. The method differs from what Barrero et al. [30] did on expansive tree grammars. Their approach, too, aims at identifying equivalent nonterminals and, additionally, reducing them. But they do not compare accept/reject behavior. Instead, they compare tree branches themselves. From the equivalence checking, a theorem follows that indicates that the languages are equivalent if the parse trees for two grammars are equivalent. According to Fischer et al. [13], matching nonterminals is possible for grammars that are not fully equivalent. The result of nonterminal matching can be used as a starting point to converge two grammars by corresponding transformations.

The testing approach to grammar equivalence is limited in the sense that it may find a counterexample for equivalence. In the case of finding one, the result is clear. But the approach cannot prepare a definite conclusion, only provide evidence, if no counterexample occurs as the problem is undecidable.

### 2.2.3 Related Work

In this subsection, we take a look at related work. In various settings, researchers and engineers may ask themselves whether the models, programs, or languages they are dealing with are equivalent. We consider a case of equivalence in process models as a representative for work in that research field. Furthermore, we visit the closely related areas grammar recovery, especially grammar inference, and language documentation.

**Equivalence in Models**

Gerth et al. [31] examine how to detect equivalence in business process models. They decide to compare models at a semantic level to bypass different syntactic notations. As a means to handle the different models, they define a normal form. They split the models under investigation into fragments, which they normalize. In an automated way, the process models are translated into process model terms. These terms are normalized using a rewriting system and then checked for equivalence. Comparing the fragments allows them to draw a conclusion concerning equivalence and builds a basis for merging the models if wanted.

**Grammar Recovery**

Language convergence often involves grammar recovery. In the cases above, grammars have been recovered from various sources and examined for equivalence. But different approaches exist to
recover a grammar. The approaches themselves can lead to a unified grammar for at least a
subset of sources. One approach is grammar inference.

Saha and Narula [32] have developed a system called Gramin. The system infers a grammar
from sample programs using grammar inference techniques. They develop a backtracking algo-
rithm that iteratively learns the grammar behind the sample programs. The system can also be
used to complete a recovered and possibly converged grammar. It can serve as an extension to a
framework that recovers and converges grammars if completion is needed. Via certain goodness
criteria, the system decides which rules to keep and which to drop if they contradict or which
new rules to add for completeness. Dubey deals with the question of suitable goodness criteria
for grammar rules [33].

Language Documentation

As the grammar invariably differs depending on its purpose, a framework can help in various
parts. It cannot only check whether the grammars converge as described above, but it can
also be used to generate one grammar from the other. Zaytsev claims, the process makes their

2.2.4 Future Work

Work concerning language convergence includes investigating programming languages for con-
vergence that have not been examined so far. To this end, the set of grammar extractors has to
be extended as well. On a lower level, it may be beneficial to incorporate precedence rules either
by accounting for precedence during abstraction or by first transforming the grammar such that
precedence rules are encoded in the grammar.

A topic pure language convergence excludes is what to do with the result. In a smaller scale,
it is possible to align the different grammars to build a consistent foundation. If the evolution
of a language is monitored by language convergence, there may be more steps to automate,
for example, adapting parsers, compilers, or documentation to accommodate changes in the
grammar.

To look at a completely different approach, it may be worthwhile comparing grammars on a
semantic level like the process modelers. Research concerning the idea may already exist but is
not part of this overview.

2.2.5 Summary and Further Procedure

In summary, most research activity in the area of language convergence circles around Zaytsev,
Lämmel, and their affiliates. So far, an operator suite for grammar transformations, frameworks
for language convergence, and grammar comparison based on generated test data exist. Future
work is manifold concerning new languages to test or refining the existing tool chain.

We will use Zaytsev’s approach of using a meta language to compare our Modelica sources.
His work includes a case study of the programming language Java. The case study involves
the convergence of three different versions of the specification and implementation of the Java
language. The comparison of a specification and an implementation is what we aim at with the
Modelica language. As a result of Zaytsev’s work, an extensive framework for convergence and
transformations exists we can utilize and rely on. In the next chapter, we delve into the details
of this framework for comparing the three Modelica grammar versions.
Chapter 3

Language Convergence Infrastructure

The tool “Language Convergence Infrastructure” (LCI) is our choice to approach the comparison of the different Modelica grammar versions. It helps to identify differences in grammar knowledge of various software artifacts. The Modelica grammar sources are the language documentation and parser descriptions. The method has first been published by Vadim Zaytsev and Ralf Lämmel[10] and is continuously developed further (e.g., see [33], [11], [12], [35]). We already had a brief look at their work concerning language convergence in the previous chapter. Now, we recapitulate some of the information we have gathered during this brief look and inspect the core tool and its components in more detail.

In contrast to other work in the area of comparing grammars, the developers built a generic tool, not only applicable to certain domains, parser descriptions, or syntax definitions. A grammar engineer can use it to verify the relationship between grammar knowledge from various software artifacts. The tool uses a unified grammar format as a representation for the knowledge, called BGF. The BGF facilitates the analysis of various forms, XML schemas, object models, etc., thus making it flexible. Apart from verifying correspondence between grammars, the tool also allows for version control between grammar versions. While a grammar evolves over time, the tool helps to track the changes or verify a backward compatibility of the latest version of a grammar with its predecessors. The changes we are able to track are limited to those still in the grammars after extraction.

The outline for this chapter is the following. We start with an overview of the tool in Section 3.1. In Section 3.2, we explain how to set up the tool, including any prerequisites and how to invoke the tool. We continue with a closer look at the infrastructure of the tool, more precisely, the different phases the developers have divided the process into, in Section 3.3. The framework for the grammar transformations is the topic of Section 3.4. We end with a small example of running a LCI in Section 3.5.

3.1 Overview

Grammar convergence is a verification method. The idea is to extract the grammar from software artifacts and transform them until they become syntactically identical.

To that end, Zaytsev and Lämmel have developed the grammar format BGF. It adopts the expressiveness of an EBNF, but also permits other constructs, including simple types and namespaces. It is expressed in XML. A version of the grammar for BGF, without namespaces and a type system limited to integers and strings, is shown in Appendix B.1. The decision in
CHAPTER 3. LANGUAGE CONVERGENCE INFRASTRUCTURE

favor of an own grammar format allows for a neutral representation impartial to any specific
models or semantics.

The first step in the convergence process is to extract the grammars. A grammar extractor
is a tool that is most likely defined in the computational framework of the artifact. It parses
the source artifact and produces a grammar in BGF. The next step encompasses a grammar
transformer and a grammar comparator that take the previously generated grammars as input.
The transformer applies grammar transformations and the comparator compares the grammars.
This transform-and-compare cycle typically ends when the input grammars become syntactically
identical.

The grammar transformations are part of a framework set up by the developers. The transfor-
mations form the language XBGF, where X stands for transformation. The framework consists
of partially automated grammar transformations that refactor or edit grammars. The term
refactoring refers to semantics-preserving transformations whereas editing covers non-semantics-
preserving transformations. The advantage of the framework is that the transformations have
been checked for correctness.

The grammar convergence framework is publicly available. It includes grammar extractors for
various programming frameworks, the grammar transformation framework, as well as the gram-
mar comparator for comparing grammars and the transformer for applying transformations from
the XBGF framework. A language documentation for the grammar transformation framework
is also available.

3.2 Setup

The LCI is part of the SLPS as mentioned before and as such can be downloaded from its
repository. A fail-safe way to get all the necessary scripts is to check out the whole suite though
only a portion of the files is needed. Checking out the whole project and familiarizing yourself
with the project structure has the advantage that all the paths in the scripts that refer to other
scripts are still valid and need not to be updated.

The scripts are written in Python to run on a Unix system. For the tool to work, a specific
Prolog parser is needed, SWI Prolog. At the end of the LCI, diagrams of the convergence
process are generated, which are set in Dot. If the diagrams should be converted into pdf files
automatically, a program to handle the diagrams must be installed. Not having such a program
installed, however, will not prevent the tool from working.

For the tool to properly function, it might be necessary to exchange a line in two scripts. These
scripts are lci.py and lciConfig.py, located in the SLPS at topics/convergence/lci. Both
scripts include the line from elementtree import ElementTree. Depending on the structure
of the XML installation in the operating system, this line needs to be changed into the expression
from xml.etree import ElementTree.

The tool is invoked by the following line:

$ ./lci <input>.lcf arch

The first argument is the configuration file that makes the paths, sources, scripts, and transfor-
mations known to the LCI. The configuration file has to be in LCI Configuration Format (LCF).
LCF is a configurational domain specific language used by the LCI. The LCF file is a XML
document specifying where the scripts, the sources, and transformation files are located. The
grammar of the LCF is in EBNF with selectors. For example, the selector a refers to a particular
occurrence of a nonterminal b by writing a::b. The built-in symbols of the LCF are string

\footnote{SLPS repository: https://github.com/grammarware/slps}
3.3 LCI Process

For any string, $xstring$ for a macro expanded string, $id$ for a unique identifier, and $refid$ as a reference to an $id$. Appendix C.1 shows the complete grammar. According to the grammar, the scenario needs at least one source and one target, a transformer to apply transformations, and a comparator to compare the sources if there is more than one source. The second argument specifies the name under which the diagrams are saved at the end of the program run.

The comparison executed by LCI is trivial. It is restricted to string comparison supplemented by a handful of normalizing rules. Before comparing the input grammars, the script normalizes the grammars by eliminating duplicate choices, dissolving sequences of sequences or choices, and handling empty elements. The normalization is also performed after the application of transformations. The single productions with the heads and bodies are identifiable within the BGF by a script. The script then checks whether matching heads exist and if they do, whether the bodies agree. Non-matching heads are exclusive nonterminals for the corresponding grammar and mean a nominal mismatch. Matching heads with diverging bodies are so-called structural mismatches. The command line output keeps up with the progress of the process and the success of each step along the way. The tool produces a result file that lists the common nonterminals with different definitions.

The LCI invokes the comparator script to compare the input grammars. Calling the comparator script directly with the two input grammars in BGF, a more detailed output is given. Additionally, the exclusive, defined nonterminals are listed as well as the colliding production bodies. The production bodies are given in prefix notation with a semicolon denoting a choice, a comma specifying a sequence, a star, plus, or question mark describing a star, plus, or optional expression, respectively, and the letters ‘n’ and ‘t’ characterizing a non-/terminal. The output helps to identify transformations to align the grammars in a stepwise manner.

We proceed with the LCI process, i.e., the inner structure of the LCI and a guideline to the transformation programming.

### Figure 3.1: Abstract Convergence Process

![Diagram](https://via.placeholder.com/150)

Source grammar 1

Transformation

Source grammar 2

Transformation

Target grammar

If two grammars are under investigation, an initial comparison of the grammars is reasonable. Running the script at the beginning reveals the overall differences between the given sources. After the initial run of the LCI, the grammar engineer can start to program grammar transformations to align the sources. The transformations are directed towards a target, a limit. In a scenario where two grammar sources belong to a target, one source is usually determined to be the objective to be reached. The other source is subject to transformations, which transform the source into the objective.

Figure 3.1 illustrates the abstract process of language convergence. Two source grammars are transformed to meet the target grammar, the common denominator between the two grammars. The target grammar can be a source grammar in a new transformation process, which turns the
CHAPTER 3. LANGUAGE CONVERGENCE INFRASTRUCTURE

target into an intermediate target. By stepwise converging the different grammar sources in a compare/transform cycle, all grammar sources are transformed to the grammar limit, the final target.

To provide more structure to the convergence process, five phases form the process of which each but the first is optional, namely

- **extraction** of grammars from different sources/artifacts,
- **convergence preparation**, e.g., fixing known errors,
- **nominal matching**, e.g., rename, inline, unchain,
- **structural matching**, i.e., semantics-preserving refactorings, and
- **resolution** (extension, relaxation, correction).

From one phase to the next, the number of differences between the grammars must decrease. The first two phases are executed once per source.

In the first phase, a script is responsible to extract the grammar from its source and transform it into the internal representation format BGF. For each grammar source, a script must be specified. The LCI saves the extracted grammars at certain paths to be able to fall back on them whenever an error occurs or the process restarts. In the next phase, a script can correct known or obvious errors. During nominal matching, any renaming, (un)folding, or other semantics-preserving transformation applies to dissolve nominal mismatches. The phase is followed by structural matching, which includes any semantics-preserving grammar transformations that align productions with structural mismatches. The last phase, resolution, refers to grammar transformations that are not semantics-preserving. They can be semantics-increasing or -decreasing. The phase is divided into three parts, extension, relaxation, and correction. During extension, grammar differences are resolved that can be explained by language evolution. Relaxation addresses the issue that grammars used for implementation are usually more permissive than grammars optimized for readability. The last part centers around the remaining differences. These differences are not meant to be and handled as a newly found bug and therefore, need correction.

Only after all semantics-preserving transformations are exploited and the sources cannot be aligned further, the grammar engineer moves on to the resolution phase. The grammar transformations programmed during this phase should be considered carefully, as they alter the scope of the grammar. Moreover, no zig zag transformations should be done, meaning that one transformation increases the semantics while another transformation decreases the semantics. The transformations might cancel each other out and the statement about a relation between the source grammars can become more vague.

The transformations programmed for the LCI are in XBGF, the grammar transformation language set up specifically for this purpose. XBGF is XML-based as well. The transformer takes a transformation written in XBGF as an input as well as the grammar to which the transformation should be applied and generates the transformed grammar. We familiarize ourselves with XBGF in the upcoming section.

3.4 Programmable Grammar Transformations

The transformation language XBGF provides 49 transformation operators. They are categorized by their effect on the semantics of an expression, i.e., semantics-preserving, -increasing, and -decreasing. Some of them are fully automatized, others need the specific expression that is to be transformed and what it should be transformed into. Then, the transformer searches for the specific expression and replaces it with the new one. For the comparison of the Modelica versions, we will predominantly use the following transformation operators:
3.5. EXAMPLE LCI

Listing 3.1: Example Productions

[Modelica Specification] subscript :
";:" | expression

[JModelica] subscript :
"COLON" | exp

- renameT, renameN
- inline, extract
- massage
- factor, distribute
- vertical, deyaccify

The first item refers to renaming terminals and nonterminals. Inlining a production replaces every instance of the nonterminal with its definition and eliminates the production. If we extract a production, a new nonterminal is specified with its defining production body. Every occurrence of the production body is then replaced by the new nonterminal. The operator massage allows rewriting expressions according to certain equivalence rules, like a double optional equals an optional or an optional equals a choice of an epsilon and the option, as well as distributivity laws. The operator distribute dissolves inner choices by distributing the sequences over the inner choices. The operator factor pulls an identical expression out of choices. The next item refers to a transformation that breaks up all top-level choices into individual productions. In the cases of the operators extract, massage, and factor, the scope can be limited from the whole grammar to a nonterminal or one production of a nonterminal. The argument of the transformation operators distribute and vertical represents the scope of the transformation. Deyaccification refers to procedures where recursive definitions are converted to iterative definitions. The transformer checks whether the grammar productions for the specified nonterminal fulfills a yaccified pattern. If true, the expression is replaced using regular expressions. The transformation can only be applied to two vertical productions of one nonterminal. The demand for vertical productions leads to the transformation often occurring in combination with the transformation vertical.

The last section before the application of LCI to Modelica deals with a simple example of the LCI. The example is taken from the Modelica grammar versions.

3.5 Example LCI

After concentrating on the structure and the ingredients of the LCI, a small example shows a practical application. We consider an expression from the Modelica specification and from JModelica, which presumably model the same construct, a subscript; see Listing 3.1. The JModelica production is devoid of the parser-specific code. For the time being, we convert the two grammar excerpts by hand into BGF. Doing so allows us to become familiar with the structure of a BGF. In both cases, the grammar consists of one production with the head nonterminal subscript and a body expression that is a choice of two expressions, one a terminal, the other one a nonterminal. This description of the grammar excerpts directly translates into the BGF description. Each grammar consists of one production with a nonterminal and an expression. The expression comprises an XML tag choice. Two expressions compose the choice, one being a terminal, one a nonterminal. With the help of the BGF grammar in Appendix B.1 we construct the BGFs by deriving the description and writing XML tags for each label or nonterminal we derive. The resulting BGFs are shown in Appendix B.2.
We start introducing the different components of the initial LCF for the LCI of the two subscript expressions. The following lines open and close a LCF file.

```xml
```

The first component are shortcuts.

```xml
<shortcut>
  <name>impl</name>
  <expansion>../</expansion>
</shortcut>
<shortcut>
  <name>s src s</name>
  <expansion><expand>impl</expand>/srcs</expansion>
</shortcut>
<shortcut>
  <name>extraction</name>
  <expansion><expand>impl</expand>/extraction</expansion>
</shortcut>
<shortcut>
  <name>tools</name>
  <expansion>../../slps-master/shared/tools</expansion>
</shortcut>
```

The shortcuts define paths to different positions in the file system where the scripts and grammars are stored. They facilitate navigating through the system. The first shortcut, for instance, names the path to the root directory from the point where the LCF is stored.

The next part defines the tools for the process.

```xml
<tools>
  <transformer>
    <grammar><expand>tools</expand>/xbgf</grammar>
  </transformer>
  <comparator>
    <grammar><expand>tools</expand>/gdt</grammar>
  </comparator>
</tools>
```

For the process to work, a transformer and a comparator are mandatory. They are specified by their path.

The next part are the grammar sources, which need at least a name and a grammar source.

```xml
<source>
  <name>leaf−expr−j</name>
  <grammar>
    <extraction>
      <expand>extraction</expand>/j2bgf <expand>srcs</expand>/jmod/leaf−expr−j.txt
    </extraction>
  </grammar>
</source>
<source>
  <name>leaf−expr−m</name>
  <grammar>
    <extraction>
      <expand>extraction</expand>/m2bgf <expand>srcs</expand>/mmod/leaf−expr−m.txt
    </extraction>
  </grammar>
</source>
```
In the example, there are two different sources, each having a unique and descriptive name. The grammar source is specified by an extractor and the source file. For the example, the extractor does nothing but copying the source as the source already is in BGF. We do so because the LCI requires an extractor and with that also saves the newly generated BGF files under certain paths. Writing a script that creates a new file to which it copies the contents of the other file allows the LCI to save that file under the path the LCI needs.

The final part is the target.

```xml
<target>
  <name>node</name>
  <branch>
    <input>leaf−expr−j</input>
  </branch>
  <branch>
    <input>leaf−expr−m</input>
  </branch>
</target>
```

A target needs an identifier and at least one branch. A branch has an input, i.e., a grammar source or intermediate target. Optionally, a branch has phases in which given transformations are applied to the input. In the example, the target under the name `node` has two branches with the two grammars as inputs. No transformations are given as we do the first comparison of the sources. Listing C.2 in Appendix C.2 shows the LCF as a whole.

If we invoke the LCI with the LCF for the subscripts, we get the following output.

```
Read 5 shortcuts, 2+0 tools, 1 targets, 2 sources, LCF is fine.  
[ PASS ] Extracted a newer version of lead−expr−j.bgf.  
[ PASS ] Extracted a newer version of lead−expr−m.bgf  
[ PASS ] Extraction finished.  
[ FAIL ] Mismatch in target node: leaf−expr−j.bgf differs from leaf−expr−m.bgf  
[ PASS ] Target node reached as leaf−expr−j.bgf  
Grammar convergence phase finished.  
------  
[ WARN ] No testing performed.  
[ WARN ] Detailed diagram not generated  
[ PASS ] Diagram generation completed.  
```

The first line is a status line about the LCF, telling us that the LCI encountered two tools, which are the comparator and the transformer, one target, and two sources. The next three lines show that the grammar sources have been extracted. Even if the extraction is executed the first time, the LCI states in its output that it extracted newer versions of the grammars. The following line reveals the results of the comparison of the two sources. One branch is then saved as the target. The convergence phase is complete with the saving of a target. The last three lines do not have to concern us. The two warnings turn up because, first, we do not specify test cases, and second, part of the diagram generation is turned off by default.

If we call the comparator script directly with the two sources as input, we get the following output.

```
Normalizing mod/leaf−expr−m.bgf.  
Normalizing jmod/leaf−expr−j.bgf.  
Differing mod/leaf−expr−m.bgf and jmod/leaf−expr−j.bgf.  
   Names of defined nonterminals agree.  
   Comparisons per (common) nonterminal:  
   - Fail (1/1): subscript.  
     - []::(t();n(expression))  
     vs.  
     - []::(t(COMMA),n(exp))  
   - Roots agree.  
```
We now have one definition below the other of each colliding nonterminal, in our example just one. We can see that we have to rename the terminal and the nonterminal. Renaming terminals and nonterminals follows the same pattern. We have to specify the name of the term we want to rename and to what it should be renamed. When the transformer applies the transformation, every occurrence of the term is replaced by the new one. The pretty-printed versions of the example transformations are `renameT("COLON", ";");` and `renameN(exp, expression);`. The XML description is straightforward. We indicate that we want the transformation `rename` followed by either a terminal or a nonterminal tag depending on what we want rename. Within this tag, we specify from what we want to change the term to. The resulting XML including the start and end tags looks like the following.

```
<xbgf:sequence xmlns:xbgf="http://planet-sl.org/xbgf">
  <xbgf:rename>
    <terminal>
      <from>COLON</from>
      <to>:</to>
    </terminal>
  </xbgf:rename>
  <xbgf:rename>
    <nonterminal>
      <from>exp</from>
      <to>expression</to>
    </nonterminal>
  </xbgf:rename>
</xbgf:sequence>
```

Next, we update the target in our LCF by a reference to this transformation. The target is then defined via the following description.

```
<target>
  <name>node</name>
  <branch>
    <input>leaf-expr-j</input>
    <structural-matching>
      <perform>rename-j-leaf</perform>
    </structural-matching>
  </branch>
</target>
```

The updated LCF can be seen in Listing C.3 in Appendix C.2. If we run the LCI again, we receive the following output.

```
Read 5 shortcuts, 2+0 tools, 1 actions, 1 targets, 2 sources, LCF is fine.
[PASS] Extraction finished.
[PASS] Applied structural matching rename-j-leaf.xbgf to leaf-expr-j.bgf
[PASS] Postextraction and synchronization finished for target node.
[PASS] Branch finished as leaf-expr-j.rename.bgf
[PASS] Target node reached as leaf-expr-j.rename.bgf
```

---

Grammar convergence phase finished.

```
[WARN] No testing performed.
[WARN] Detailed diagram not generated
[PASS] Diagram generation completed.
```

The LCI has found one action, the XBGF transformation, in the LCF. In comparison to the previous LCI run, one line provides the status of the extraction phase. With the first LCI run, the sources have been extracted and the files saved. The LCI can go back to these files and does not need to call the extractor again. The output contains a line informing us about the successful
application of the transformation. The next two lines brief us on the status of the comparison, which was successful. The remainder of the output is identical to the aforementioned output.

The convergence process is visualized in Figure 3.2. The figure belongs to the diagrams generated by the LCI. The last line of the output informed us about the successful completion of the diagram generation. If the input branches do not meet the target, the LCI generates a diagram where the target is typeset in a red color.

With the example, we have seen how the LCI works in small scale. In the next chapter, we tackle a larger set of grammar productions as the subjects of the LCI.
Chapter 4

LCI for Modelica

In this chapter, we use the LCI to compare our three Modelica grammar versions. The purpose of this chapter is twofold. One, we will see how to use the LCI to compare and transform the three grammar versions. Two, we collect evidence for a statement concerning the relation between the grammar versions and their equivalence.

The outline of the chapter is the following. We start with a brief look at preceding choices about how to set up the LCI in Section 4.1. The choices include what parts of the LCI we use and what the grammar is we choose to transform the other grammar towards. The next section, Section 4.2, centers around the extraction of the three grammar versions and conversion into BGF. We continue with a description of the work on the pairwise comparison of the grammar versions. Section 4.3 describes the comparison of the Modelica specification with JModelica, Section 4.4 the comparison of the specification with OpenModelica, and Section 4.5 the comparison of OpenModelica with JModelica. The three sections all follow the same structure of an initial comparison, the grammar transformations, and a look at the final results, with a summary at the end. We will see that actually the grammar versions do not converge but exhibit differences that do not lead to a distinct relation between the versions. The first part, the Modelica specification vs. JModelica, receives a more detailed explanation to clarify the process. The other two parts will be handled in a more condensed way. At the end of this chapter, we briefly summarize our work (Section 4.6). An evaluation of the results will take place in the next chapter, Chapter 5.

4.1 Preliminaries

For the comparison of the Modelica grammars, we use a “core” LCI, specifying only a comparator and a transformer. The three grammar versions are the sources within the LCI. For this thesis, we restrict the grammars to expressions. Starting point for deciding which productions to consider is the Modelica specification. We include the productions in the section “Expressions” in the Modelica specification. The definitions of referenced nonterminals outside this section are ignored. We compare two sources at a time, thus, the three pairings separately. We stop after the application of semantics-preserving transformations. We do not move on to the last phase of resolution as we are interested in the differences and not a full language convergence. The objective of the convergence process is the Modelica specification in the first two cases as it represents the original grammar as published by its creators. For the comparison of the grammar versions from OpenModelica and JModelica, the OpenModelica grammar becomes the target. The choice is fairly arbitrary, the motive in this case being that OpenModelica’s grammar structure resembles the one of the Modelica specification more than JModelica’s grammar structure.
With the sources and the target as well as the comparator and the transformer, we have the main components for the LCI. The next step in the work process is to write the scripts we need to transform the sources into BGF, which is subject of the upcoming section.

4.2 Grammar Extractors

For each grammar version, we need a script to transform the grammar from its original form into BGF. Doing the transformation by hand is very cumbersome as the representation in BGF is rather complex and long even with only part of the grammar. The scripts are written in Python as are most of the scripts in the LCI. The scripts handle the expressions of the respective grammars and any other simple input without nested choices or highly nested star, plus, or optional expressions. In all three cases, a preprocessing takes place to prepare the grammars for transformation into BGF. The preprocessing includes deleting the parser-specific code and printing each production in one line. The main transformation script goes through each line of the file and prints the production accordingly. The preprocessing script takes the original grammar version as an input, and the transformation script then takes the file generated by the preprocessing script.

The body of this section consists of a subsection for each grammar version where we take a closer look at the scripts.

4.2.1 Modelica Specification

The Modelica grammar is available in its language specification document. The section for expressions is copied into a text file, which serves as the input for the preprocessing script. The productions have a similar structure, which is used as a guideline to build the script. Terminals are enclosed by quotation marks to distinguish them from the nonterminals. As the Modelica keywords are marked in bold in the document, a property that is lost during copying, we need the script to mark them with quotation marks as well. In most cases, the production head is in one line and the body, depending on its length, in the following lines. The head and the body are further separated by a colon and the body is indented. The first production of the section “Expressions” in the Modelica specification grammar showcases this behavior:

```
expression :
    simple_expression
    | "if" expression "then" expression { "elsif" expression "then" expression } "else" expression
```

The preprocessing script reads each line of the text file. If the line starts with a non-whitespace character, a new production starts. The production head, the first word of such a line, is printed into the new text file. The colon character is not printed in the new file. If the line ends there, the script goes on to the next line. If the line continues, part of the production body is in the same line as the head. Then, the line is split after the colon, and the latter part is saved for the printing of the production body. If the line starts with a whitespace character, we save the lines in a list until the script encounters another line with a non-whitespace character, which means that the production body is complete. Before handling the new production, the body is printed into the new file. After preprocessing, the production above looks like the following:

```
expression simple_expression | "if" expression "then" expression { "elsif" expression "then" expression } "else" expression
```

The procedure repeats until the script reaches the end of the file. Then, the last production is printed and the files are closed.
The transformation script prints the headers of the XML file and then goes through each line of the file and prints the productions. The first word is the production head, the remainder of the line the body. The head is printed, and a function to print the body is called. According to the length and the types of markers like a vertical bar for choices, curly braces for star expressions, and square brackets for optional expressions, the respective XML expressions are printed and further functions are called to print the inner parts or choices. The printing and calling of functions continues until we are left with single terms, i.e., terminals and nonterminals, to print. At the end, the concluding expressions for the file are printed and the files closed.

Both scripts are combined in an executable that handles the input and output files. The executable needs the input grammar and the output file for the BGF at the end as input arguments. The executable is also the one specified in the LCF file as the extractor. The LCI takes care of feeding the executable with the input.

4.2.2 JModelica

JModelica specifies the grammar in the parser description. As we restrict the Modelica grammar to the expressions section, we take only the corresponding productions from the JModelica grammar. To transform the grammar into BGF, we need to delete the parts that are parser-specific and do not carry any information concerning the grammar. For JModelica, this parser-specific code is enclosed by {: and :}. The preprocessing script strips the grammar off these parts. The script also deletes class and variable names. The end of a production is marked by a single semicolon in a line. The corresponding production to the production in the previous subsection reads in JModelica:

\[
\begin{align*}
\text{Exp exp} &= \text{simple_expression} \{ : \text{return simple_expression; :} \\
& \mid \text{if_exp} \quad \{ : \text{return if_exp; :} \\
& ;
\end{align*}
\]

The script follows the same procedure as the one for the Modelica specification. It goes through each line of the file. If the line starts with a non-whitespace character, a new production starts. As we have class names, the nonterminal that is the production head is the second word in the line. The body of the production is appended to a list line by line until the end symbol is read. While appending the production body, the parser-specific parts and variable names are omitted. The body is printed in a line with the production head. The script converts the production given above into the following line:

\[
\begin{align*}
\text{exp} \text{ simple_expression} \mid \text{if_exp}
\end{align*}
\]

One property of the JModelica grammar is that no nested choices, star, or plus expressions exist. Additionally, optional expressions are only expressions with one term, not sequences. Therefore, the script can go through each line of the file, print the first word of the line as the production head and call a function for the rest of the line. If there is a vertical bar in the line, we have a choice and can print the needed XML expressions. For each choice, another function is called, which also handles the body if no choice is at hand. After this point, each word can be handled as a single term testing whether it ends with a question mark for an optional expression or is uppercase for terminals. Else, we have a simple nonterminal. For each case, the function prints the needed XML expressions. The script also prints the starting and ending expressions for the XML format. The result is the JModelica grammar in BGF format.

As with the Modelica specification, an executable manages the calling of the scripts with the correct inputs and represents the extractor in the LCF file.
4.2.3 OpenModelica

The OpenModelica grammar is embedded in the parser description generated by ANTLR. Again, some preprocessing is necessary to delete the parser-specific code. The parser-specific code is either marked by curly braces or a return statement beginning directly after the production head and optionally followed by a declaration part introduced with @. The return statement closes with a colon, which marks the end of the head and the beginning of the body of the production. As with JModelica, the parser includes variable names that need to be deleted and the production body must be accumulated to be printed in one line. The script does the preprocessing in the same fashion as the other two scripts, producing a text file with one production per line and the head as the first word. The production

```plaintext
expression returns [ void* ast ]:
  ( e=if_expression { ast = e; }
  | e=simple_expression { ast = e; }
  | e=code_expression { ast = e; }
  | e=part_eval_function_expression { ast = e; }
  | e=match_expression { ast = e; }
  )
```

transforms to the following line:

```plaintext
expression if_expression | simple_expression | code_expression | part_eval_function_expression | match_expression
```

The transformation script resembles the one for the Modelica specification as the OpenModelica grammar is similar in structure to the specification grammar. The vertical bar represents a choice. Optional expressions are marked by a question mark, star expressions by a star. Terminal nodes are identifiable by checking whether a term is uppercase. For a description of the approach to print the grammar in BGF, please refer to the Modelica specification extractor description in Section 4.2.1. The same holds for the executable that is referenced in the LCF file as the extractor.

With the extractors for the grammar sources, we can proceed and start to compare the grammar versions. We will start with the comparison of the Modelica specification with JModelica.

4.3 Modelica Specification vs. JModelica

The two grammars subject to comparison are the Modelica specification and JModelica. The first step in this section is to write the LCF file to start the comparison with. Without any transformations specified, the LCI outputs the differences between the two grammars. After the first run of the LCI, we start to analyze the differences and program transformations to align the grammars. After no semantics-preserving transformations can be applied anymore to further dissolve any differences, we investigate the remaining differences. The differences involve the logical and arithmetic expressions, names and single function arguments, which are more relaxed in JModelica. The constructs where the Modelica specification is more permissive are the component references, the primaries, more precisely, the built-in functions `initial()` and `der(...)`, the output expression lists, and the list that is allowed before an expression involving the keyword `for`.

Each of the following subsections delves into one of these working steps. First, we take a look at the initial LCI run, with its LCF and the output in Subsection 4.3.1. Next, we go through the transformations that are targeted towards resolving the differences identified in Subsection 4.3.2. Last, we analyze the output of the LCI with the transformations of the preceding subsection specified in the LCF file and thus, applied to the grammar in Subsection 4.3.3.
4.3.1 Initial LCI

As the LCI takes a LCF as an input, we need to write a LCF file specifying the paths to the folders where the different scripts can be found, the paths to the sources, and the target. Listing C.4 in Appendix C.3 shows the initial LCF file. The LCF from the small example in the previous chapter (Section 3.5) has been modified to address the new sources and the new target. The sources are the Modelica specification and JModelica grammar text files with a fitting reference name. The extractors are the ones presented in the previous section. The target gets a name and the input branches, which are the reference names for the two grammar sources. As mentioned before, no transformations appear at this stage, which means that the LCF is complete.

Running the LCI with this LCF file as input delivers an output that says that differences between the two grammars exist and that the common target could not be reached. The result file tells us that eleven shared nonterminals have different production bodies. If we call the comparator script alone, we get a more detailed output that contrasts the colliding definitions. The comparator output also names the nonterminals that are used exclusively by one grammar. Listing D.1 in Appendix D.1 shows the comparator output. The output reveals that the terminals all differ as JModelica uses uppercase letters and writes out symbols. That is a starting point for transformations, renaming the terminals to make them identical. The same holds for some nonterminals where it is obvious that they are meant to be identical, e.g., expression and exp. For the other differences, we have to take a look at each and may take other productions into consideration that are not part of the fail note. Stepwise, we work ourselves through the list of failed productions and try to include the nonterminals not shared by now.

The next subsection examines the transformations applied to the JModelica grammar.

4.3.2 Transformations

This subsection examines the transformations applied to JModelica. The transformations form two groups. To the first group belong the transformations where the productions actually converge. The second group consists of transformations that do not lead to identical productions. But in order to narrow down the differences or to even be able to identify the differences, we program the transformations nonetheless. We will first go through the transformations where the targeted productions converge and then through the transformations of diverging productions. Appendix E.1 lists all transformations that are applied to JModelica in the process of comparing JModelica and the Modelica specification.

Renaming Terminals and Nonterminals

We have visited the renaming of terminals and nonterminals in the LCI example in Section 3.5. We do the same with the Modelica specification and JModelica expressions as sources. We gather the differing terminals and nonterminals that need to be renamed to nominally match and program the corresponding transformations. The renaming leads to the productions annotation and subscript to become identical. Later in the process, additional renamings of nonterminals will take place as in some cases, it is not obvious from the beginning which the corresponding Modelica specification nonterminal is to which the JModelica nonterminal needs to be renamed.

Subscripts

The first two fail notes in the output list relate to the production array_subscripts. The JModelica production uses another production in this context, namely subscript_list. We transform the JModelica production such that it becomes identical to the Modelica production.
First, we deyaccify the production \texttt{subscript\_list} to strip the production of the recursion and second, inline the production \texttt{subscript\_list}. The transformations change the productions in the following way:

\begin{verbatim}
subscript_list : subscript | subscript_list "," subscript
// vertical( in subscript_list ); deyaccify(subscript_list);
\Rightarrow subscript_list : subscript ( "," subscript )
array\_subscripts : [" subscript\_list "]
// inline(subscript_list);
\Rightarrow array\_subscripts : "[" subscript ( "," subscript )"]"
\end{verbatim}

The part after the two slashes lists the pretty-printed transformation applied whose result is shown in the next line. The last line equals the Modelica definition. The transformations, \texttt{deyaccify}, \texttt{inline}, and \texttt{vertical}, which is needed for \texttt{deyaccify} to work, are automated transformations, which are handled by the framework and only need the nonterminal to be specified to which the action should be applied.

\textbf{Expressions}

Going on to the next fail note, we see that the Modelica specification defines an expression via a simple expression or a sequence of terminals and nonterminals generating an if expression. The JModelica production calls for another nonterminal, \texttt{if\_exp}. If we consult the productions \texttt{if\_exp} and the referenced production \texttt{elseif\_exp}, we can see that the same transformations as for aligning the subscripts apply to align the production.

\textbf{Simple Expressions}

The previous production referenced the production \texttt{simple\_expression}, which is also subject of the next fail note in the output list. The difference between the two colliding productions is that one uses choices to express three different possibilities of production whereas the other has a nested optional expression. We can turn the first into the latter by factoring out the nonterminal \texttt{logical\_expression} in each choice. The JModelica version reads, with \texttt{l\_e} abbreviating the nonterminal \texttt{logical\_expression}:

\begin{verbatim}
l\_e | l\_e ":" l\_e | l\_e "::" l\_e
\end{verbatim}

A sequence of the nonterminal \texttt{logical\_expression} with an inner choice results from the transformation \texttt{factor}; the inner choice consists of an epsilon expression and the other two choices minus the nonterminal \texttt{logical\_expression} in each choice. The JModelica version reads, with \texttt{l\_e} abbreviating the nonterminal \texttt{logical\_expression}:

\begin{verbatim}
l\_e ( \epsilon | ":" l\_e | "::" l\_e "::" l\_e )
\end{verbatim}

An epsilon as a choice can be used to transform the rest of the choice into an optional expression.

\begin{verbatim}
l\_e ( "::" l\_e | "::" l\_e "::" l\_e )
\end{verbatim}

Now, we have formed the outer optional of the nested expression. The inner part of the optional expression needs to form a sequence with the beginning "::" \texttt{logical\_expression} and the ending of an inner optional with "::" \texttt{logical\_expression} again. If we apply the same transformations as before, \texttt{factor} and \texttt{massage}, we achieve the sequence. We factor out the identical beginning, "::" \texttt{logical\_expression}, and transform the choice with an epsilon expression into an optional.

\begin{verbatim}
l\_e ( "::" l\_e ( \epsilon | "::" l\_e ) )
l\_e ( "::" l\_e ( "::" l\_e ) )
\end{verbatim}

The productions are now identical.

The transformations used for aligning the production \texttt{simple\_expression} are not fully automated. Not fully automated transformation require specification of the part that should be
transformed and the transformed part. Given this information, the transformer script searches for the part to be transformed and replaces it with the new part. Of course, the transformer checks whether the transformation follows the rules for the intended transformation to secure correctness within the transformation process.

Relations

For the definitions of the nonterminal relation to align, a transformation not used so far is applicable, extract. As we can easily see, JModelica has six choices where a nonterminal arithmetic_expression resides in the first and the last position and a different relational operator in the middle. Factoring out the nonterminals at the beginning and the end results into an inner choice of the different relational operators. The aforementioned transformation extracts the inner choice as the body of a new production with the head rel_op. After the extraction, another factorization and a massaging transformation is necessary to completely converge the productions relation.

Named Arguments

The Modelica specification defines named arguments by the following expression:

\[
\text{named_arguments : named_argument (""," named_arguments )}?
\]

In order to transform the JModelica definition into the one given above, we need to make a detour first. If we factor out the nonterminal named_argument and massage the new production, we will end up with the reversed production of Modelica. The optional part would come first with a recursive call and a following comma terminal, second the mandatory nonterminal named_argument. To fix the reverse order, we first deyaccify the production, massage it according to the distributivity laws for star expressions, and then yaccify it again. The result of this series of transformations is that the second choice is in reverse order. If we factor and massage the definition now, we have aligned the two productions perfectly.

The referenced nonterminal named_argument needs two transformations for convergence, inline(named_argument_id); and renameN(function_arg_exp, function_argument);

The actual XBGF file handling the conversion of named arguments consists of the following pretty-printed transformations.

```
vertical( in named_arguments );
deyaccify(named_arguments);
massage(
    named_argument (""," named_argument")*,
    (named_argument ",")* named_argument);
yaccify(
    named_arguments:
        named_argument
    named_arguments:
        named_argument "," named_arguments
);
horizontal( in named_arguments );
factor(
    (named_argument | (named_argument "," named_arguments)),
    named_argument (EPSILON | (""," named_arguments)));
massage(
    (EPSILON | (""," named_arguments)),
    (""," named_arguments)?);
inline(named_argument_id);
renameN(function_argument_exp, function_argument);
```
Comments

The last productions to align are `comment` and `string_comment`. In the JModelica production `comment`, the first nonterminal `string_comment` is optional. In Modelica, this nonterminal is mandatory but the production `string_comment` is completely optional. To align the varying occurrences of the optional, we inline the production `string_comment` and extract it with the surrounding optional. To converge the production `string_comment`, we deyaccify it.

With these transformations, we have handled all fail notes we can make disappear. Next, we consider the remaining productions. For some productions, we will be able to extract parts of the existing productions as new productions and thus, further align the grammars. The rest of the grammar gets transformed to the point where the differences become clearly identifiable.

Logical Expressions

The productions modeling logical expressions in the Modelica specification allow for expressions in disjunctive normal form (DNF), with outer disjunctions and inner conjunctions. The single terms are logical factors, defined by an optional negation and a relation. JModelica does not restrict logical expressions to this normalized form but offers a conjunction, disjunction, or single term as a choice. The single term is a relation with or without a negation. We extract the single term choice to converge part of the grammar. The referenced nonterminal `relation` can be factored out and the newly formed choice of epsilon and the terminal `not` massaged into an optional. Factorizing the rest of the production `logical_expression` is not necessary but condenses the expression and emphasizes the difference between the two productions.

Arithmetic Expressions

Arithmetic expressions in the Modelica specification have an optional sign and a term expandable by an iteration of an additive operator and a term. The terms can have multiplications consisting of factors and multiplicative operators. Each factor is a power expression. In JModelica, the arithmetic expressions boil down to the same expression but with one difference. While the Modelica specification has one optional additive operator in the beginning, JModelica allows multiple signs in front of every power expression. A star expression with a choice between plus and minus models the sign. In the process of comparing the production, we can align two productions by extracting the arithmetic operators. The additive expression of JModelica approaches the structure of the Modelica production by inlining and deyaccifying it. If we inline the production `pow_exp` and rename the nonterminal `unary_exp` to `factor`, we set the ground to converge the multiplicative expressions. Renaming the nonterminal `multiplicative_exp` to `term` and deyaccifying the production lead to identical expressions. The newly named production `factor` can adjust to the structure of the Modelica production by deyaccification. For the transformation `deyaccify` to work, we have to combine a few transformations. As said before, we have to break the main choice into vertical productions. Additionally, the transformation only works if exactly two vertical productions, or two top-level choices, respectively, exist. Therefore, we need to factor out the nonterminal `factor` before applying the transformation `deyaccify`. Now, we can no longer align the arithmetic expressions of Modelica and JModelica.

Primary

The next production under investigation is `primary`. We start by comparing the terminals in the competing productions. JModelica has two additional choices, the terminals `UNSIGNED_INTEGER` and `TIME`. Looking at the flex definition for the terminal `UNSIGNED_INTEGER`, we can confirm
that the regular expressions for the terminals `UNSIGNED_INTEGER` and `UNSIGNED_NUMBER` agree. Thus, the terminal `UNSIGNED_INTEGER` does not carry any additional information and can be eliminated. The documentation for the time expression, which the parser creates with a reference to the terminal `TIME`, reveals that a time expression is a real number. A real number corresponds to the terminal `UNSIGNED_NUMBER` within the grammar. Therefore, we can eliminate the choice `TIME` as well. To eliminate the two choices, we string together the following transformations:

```plaintext
vertical( in primary );
removeV(
  primary:
    "UNSIGNED_INTEGER"
);
removeV(
  primary:
    "TIME"
);
horizontal( in primary );
```

After we have resolved the choices consisting of terminals, we move on to the choices that include nonterminals. The production `primary` serves as a good starting point to identify nonterminals and their productions that represent the same expressions. We see that the nonterminal `matrix` should produce expressions similar to the ones generated by the sequence with the nonterminal `expression_list`. We can see that the production `output_expression_list` will not find an identical counter production, as JModelica references the nonterminal `expression`. We can move this difference from the production `primary` to the production `output_expression_list` if we extract the single nonterminal `expression` in the corresponding choice. The productions for the nonterminal `functions_arguments` already share the same name but as they do not converge, we will take a look at them later in this section. The remaining choices have to somehow represent the same expressions and we will have to take a closer look at them together.

First, we will transform the production `matrix`, then go on to the productions concerning `function_arguments` and close with the remaining choices.

**Matrix and Expression Lists**

The production `matrix` references the nonterminal `matrix_row`, which shows promise for transforming it to agree with the production `expression_list`. We need the frequently used combination of the transformations `vertical` and `deyaccify` to align the production bodies and conclude with renaming the nonterminal `matrix_row` to `expression_list`. The same transformations are necessary to transform the production `matrix` into the form that the Modelica specification uses in the choice with the nonterminal `expression_list` in the production `primary`. If we inline the production `matrix` in the end, we have converged this part of the grammar. The exact chain of transformations, without the transformation `inline(matrix)`; in the end, is the following:

```plaintext
matrix_row : expression | matrix_row " ," expression
// vertical( in matrix_row ); deyaccify(matrix_row); rename(matrix_row,
  expression_list);
=> expression_list : expression ( " ," expression)*
  matrix : expression_list | matrix " ;" expression_list
// vertical( in matrix ); deyaccify(matrix);
=> matrix : expression_list ( " ;" expression_list )*
```

**Function Arguments**

The JModelica production for `function_arguments` references a nonterminal `arg_list_p`. We can transform the nonterminal into a sequence of one occurrence and a star expression of
the nonterminal `function_argument` and inline the production. But no semantics-preserving transformation will converge the two colliding productions as the choice involving the terminal "for" starts with a general expression and not a list of the nonterminal `function_argument`. The nonterminal `for_indices` has a production that is not within the expressions section of the Modelica specification. As decided in the preliminaries, we do not follow up on the production. The production `function_argument`, which both versions reference, can be further aligned by inlining the production `partial_function_call`. As we have not yet looked at the production `parse_access`, we have to postpone any statement concerning the equivalence with the nonterminal `name`. But we already see the difference that JModelica references the nonterminal `function_arguments` whereas the Modelica specification allows the nonterminal `named_arguments`, a subset of the function arguments.

Next, we continue with the remaining choices from the production `primary` and thus, the remaining productions within the two grammars. First, we examine the names and references of the Modelica specification versus the access expressions of JModelica. Second, we compare the expressions concerning function calls, including the so-called der expression.

### Names, Component References, and Access Expressions

We have assumed in the introduction of JModelica, when surveying its parse tree, that the access productions model component references. After a closer look to the definitions, we reason that we can transform the access expressions to echo names and component references. To this end, we have to deyaccify and massage the production `parse_access` and inline the productions `parse_access_single`, `class_access_single`, `parse_access_loc`, and `first_class_access`. A few other transformations help to clarify the differences between the two productions, which involve a transformation we have not used so far, `distribute`. This transformation distributes a term over an inner choice. Distributing the production `parse_access`, factoring out the sequence "." "IDENT", and massaging the result three times will compress the production the most and still highlight the difference between the access expressions and the reference expressions. The next step is to rename the nonterminal `parse_access` to `name` and the nonterminal `access_expression` to `component_reference`. As the production `component_reference` is solely given by the nonterminal `name`, we could unfold the definition of `name` into the definition of the nonterminal `component_reference`. We could even think of equating both productions afterwards as they have the same definition. A transformation `equate` replaces one nonterminal with the other one and eliminates the production of the first one. But as we are comparing JModelica with the Modelica specification and the specification varies the definitions for the two productions, we keep both productions and only unfold the definition of the nonterminal `name` into the definition of the nonterminal `component_reference`.

### Function Calls

The remaining productions in JModelica are `der_expression` and `function_call`. The respective nonterminals are choices in the production `primary`. Relating to the Modelica specification, these productions have to generate the same constructs as the Modelica specification choice ( `name` | "der" | "initial" ) `function_call_args` in the Modelica specification definition of primaries. After a quick investigation of the JModelica productions, we can establish that we will not be able to converge these choices. We can inline the production `der_expression`. Moreover, we can inline the production `function_call` and then extract it again without the first nonterminal as the production `function_call_args`. The new production actually converges with the specification production. But the definition of the nonterminal `primary` diverges as the keyword `initial` is missing and another nonterminal is referenced after the keyword `der`. 
4.3. MODELICA SPECIFICATION VS. JMODELICA

Table 4.1: Final Converging and Failing Productions between the Modelica specification and JModelica

<table>
<thead>
<tr>
<th>Converging Productions</th>
<th>Failing Productions</th>
<th>Exclusive Modelica Productions</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_subscripts</td>
<td>logical_expression</td>
<td>logical_term</td>
</tr>
<tr>
<td>subscript</td>
<td>arithmetic_expression</td>
<td></td>
</tr>
<tr>
<td>annotation</td>
<td>factor</td>
<td></td>
</tr>
<tr>
<td>expression</td>
<td>primary</td>
<td></td>
</tr>
<tr>
<td>simple_expression</td>
<td>output_expression_list</td>
<td></td>
</tr>
<tr>
<td>relation</td>
<td>function_arguments</td>
<td></td>
</tr>
<tr>
<td>rel_op</td>
<td>function_argument</td>
<td></td>
</tr>
<tr>
<td>term</td>
<td>component_reference</td>
<td></td>
</tr>
<tr>
<td>named_arguments</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>named_argument</td>
<td></td>
<td></td>
</tr>
<tr>
<td>comment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string_comment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical_factor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>add_op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul_op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>expression_list</td>
<td></td>
<td></td>
</tr>
<tr>
<td>function_call_args</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

With this last transformation, we have exhausted the semantics-preserving transformations. The following subsection summarizes the result of the language convergence process of the Modelica specification and JModelica.

4.3.3 Results

The transformations discussed in the preceding section are listed within the LCF as transformations to be applied to the JModelica source; see Listing C.5 in Appendix C.3. If we run the LCI with this LCF as input, we still get the statement that the target could not be reached as the grammars still differ. But we have expected that to happen as we could not completely converge the grammars. Listing A.3 in Appendix A.2 presents the transformed version of the JModelica grammar. If we call the comparator script directly, we get an output shown in Listing D.2 in Appendix D.1. The output details that the transformed JModelica grammar does not use any nonterminals exclusively anymore. The one nonterminal left exclusively within the Modelica specification grammar is logical_term. Various shared nonterminals converge while others still fail. Table 4.1 lists the converging and failing productions in a concise way.

The following paragraphs discuss each item of the failing productions list including the production exclusive to the Modelica specification.

Logical Expressions

The difference between JModelica and the Modelica specification lies in the form of accepted logical formulas. The colliding productions read:

\[ \text{Modelica} \] logical_expression: logical_term ( or logical_term )
\[ \text{Modelica} \] logical_term: logical_factor ( and logical_factor )
\[ \text{JModelica} \] logical_expression: logical_factor ( ( and | or ) logical_factor )
Modelica only allows logical expressions in DNF for whose modeling, it needs the additional production `logical_term`. JModelica accepts any sequence of conjunctions and disjunctions.

**Arithmetic Expressions and Factors**

The next minor difference is in the representation of arithmetic expressions. The difference regards the position and number of signs a term can have. In JModelica, each factor can have an arbitrary number of leading symbols, plus and minus. The Modelica specification allows one leading symbol, an additive operator, at the beginning of the arithmetic expression.

**Primary**

With the production `primary`, the differences become larger. The terminals in the colliding productions agree. The difference lays in the Modelica specification choice

```
(name | "der" | "initial") function_call_args
```

against the two JModelica choices

```
"der" "( expression ")" | name function_call_args.
```

The terminal "initial" is not part of the JModelica expressions notation. The terminal "der" is followed by a nonterminal `function_call_args` in Modelica and by an expression in parentheses in JModelica. The parentheses are also part of the function call arguments but the function call arguments then refer to an optional nonterminal `function_arguments`. As the definition of `function_arguments` can ultimately lead to a single expression as well, the difference lays in the other possibilities the production `function_arguments` allows.

**Output Expression Lists**

The choice "(" output_expression_list ")" in the Modelica specification with the definition of the referenced nonterminal is not identical to the one expression JModelica allows. The specification grammar can produce a list of expressions whereas JModelica allows only one expression.

**Function Arguments and Function Argument**

One choice in the aforementioned production `primary` is "{" function_arguments "}", which is the same in both notations. But the definitions of the referenced nonterminal differ. The generation of an expression involving the keyword `for` in the Modelica specification works in the following way:

```
function_argument function_argument" "for" for_indices
```

JModelica generates such an expression by:

```
expression "for" for_indices
```

JModelica allows only an expression in front of the sequence "for" `for_indices`. The Modelica specification references the nonterminal `function_argument`, which allows more than an expression. The specification production also facilitates a list of the nonterminal `function_argument` separated a comma before going on to the for expression.

The productions `function_argument` collide through the arguments between the parentheses. JModelica jumps back to the nonterminal `function_arguments`, while the Modelica specification only allows the nonterminal `named_arguments`, making JModelica more permissive.
Component References and Names

The last productions we contrast towards each other are the component references and names. In JModelica, a dot at the beginning of a component reference leads to an identifier followed by a dot again without an array subscript in between. The rest of the definition agrees with the Modelica specification. The specification production allows the extra array subscript after the first dot and identifier.

In JModelica, the definitions of component references and names are identical. The Modelica specification defines them differently. The component references have an optional array subscript after each identifier while the names do not. Comparing the definitions of the names, we conclude that JModelica allows subscripts at all.

4.3.4 Summary

Large parts of the two notations can be aligned with simple refactoring measures. Minor differences exist when it comes to logical and arithmetic expressions where JModelica is more permissive than the Modelica specification. With the definition of the nonterminal primary and the nonterminals referenced in there, more differences arise. The differences lay in the productions name, component_reference, function_arguments, function_argument, output_expression_list, and primary. The divergence of the production primary can be narrowed down to the choice (name | "der" | "initial") function_call_args in the Modelica specification colliding with the JModelica choices "der" "(" expression ")" | name function_call_args.

The definitions of the nonterminal function_argument and name are more permissive in JModelica. Regarding the other differences, Modelica is more permissive. Apart from the terminal "initial", the differences are cases of one grammar being more relaxed than the other, making the grammars in some parts subsets and in other parts supersets of the other but not down right contradicting.

With the comparison of the Modelica specification and JModelica finished, we move on to the comparison of the Modelica specification with OpenModelica.

4.4 Modelica Specification vs. OpenModelica

With the comparison of the Modelica specification and OpenModelica, the process starts all over again. We need an initial LCF to get first results of the comparison of the two grammar versions. Based on the results, we engineer the grammar transformations and update the LCF file. If we can no longer apply semantics-preserving transformations and decrease the differences between the two grammars, we analyze the end result with its remaining differences. OpenModelica, as mentioned during the description of the parse tree in Subsection 2.1.3 references more expressions from its nonterminal expression. This difference persists throughout the comparison. The other differences concern component references, names, named arguments, and the named arguments allowed after an expression involving the keyword for, with OpenModelica being more permissive. The Modelica specification extends the expressiveness of OpenModelica concerning for-or-expression lists, the input arguments for the built-in function initial(), and the list of arguments allowed before an expression involving the keyword for.

The topics of the upcoming subsections reflect the steps in the working process with the initial LCI in Subsection 4.4.1 the transformations in Subsection 4.4.2 and the end results in Subsection 4.4.3. By that, we maintain the structure of the description of the previous language convergence attempt.
4.4.1 Initial LCI
For the new comparison, we can reuse the initial LCF for the language convergence of Modelica and JModelica and update it. We need to specify OpenModelica as a source with its extractor and as an input branch in the target; see Listing C.6 in Appendix C.4.

Running the LCI with this new LCF as input, we learn that the grammars do not converge. The sources share twenty nonterminals whose definitions collide. If we call the comparator script directly, we get the output shown in Listing D.3 in Appendix D.2. One production already converges, comment. Similar to JModelica, OpenModelica uses another style to name its terminals, and we can align these easily by renaming them. A fail note we will not address in terms of transformations is the first one about the production expression though we will inline the if expression and part of the simple expression. No transformation can decrease the core difference between the definitions as it includes nonterminals, which reference constructs not part of the specification’s expressions. We will take a closer look at this difference in the subsection that discusses the results of the language convergence process, Subsection 4.4.3. The remaining fail notes are addressed individually in the following subsection.

4.4.2 Transformations
The transformations we apply to OpenModelica do not include any transformations we have not used in the previous language convergence process. We will mainly use transformations like massage, extract, vertical, or distribute to align the grammars. A complete list of the transformations applied to OpenModelica is given in Appendix E.2. We start the process with renaming terminals and go on to the remaining differing definitions of the shared nonterminals.

Renaming Terminals
Renaming the terminals leads to the following eight definitions of the overall 20 nonterminals already shared to agree: logical_expression, logical_term, logical_factor, factor, named_arguments, subscript, string_comment, and annotation.

If Expressions
To align the expressions that represent an if expression, we need to deyaccify the production elseif_expression_list. In advance, we massage the optional recursive call into a choice of epsilon and the nonterminal itself, after which we distribute and verticalize the production. After the deyaccification, the transformation inline replaces the occurrences of the nonterminals elseif_expression_list, elseif_expression, and if_expression with their definitions and deletes the corresponding productions. Now, the choices concerning if expressions agree.

Simple Expressions
The OpenModelica production simple_expr agrees with the production simple_expression of the Modelica specification. But the production simple_expression also exists in OpenModelica. As the production expression, which references the nonterminal simple_expression, will not converge, we will shove the additional terms of the production simple_expression into the production expression, concentrating the differences there and aligning the production simple_expression by inlining the production simple_expr. Before inlining the production simple_expr, we need to eliminate the recursion in the definition. The transformations necessary are massage, factor, vertical, deyaccify, inline, and rename. We massage the optional in the first choice into an inner choice of epsilon and the previously optional part.
4.4. MODELICA SPECIFICATION VS. OPENMODELICA

simple_expr ( ":=" simple_expression )? | "IDENT" "as" simple_expression
=> simple_expr ( ε | ":=" simple_expression ) | "IDENT" "as" simple_expression

Then, we distribute the production and factor out the nonterminal simple_expression.

=> simple_expr | simple_expr ":=" simple_expression | "IDENT" "as"
  simple_expression
=> simple_expr | ( simple_expr ":=" | "IDENT" "as" ) simple_expression

Now, we have two top-level choices and we can deyaccify the production after applying the transformation vertical. With these transformations, we eliminated any recursion in the production.

=> ( simple_expr ":=" | "IDENT" "as" )* simple_expr

Hence, we inline the production simple_expression and rename the nonterminal simple_expr into the nonterminal used by Modelica, simple_expression. With this last transformation, we have aligned the productions simple_expression.

Operators

OpenModelica, similar to JModelica, does not have own productions for its relational, additive, and multiplicative operators. If we extract these operators, the productions relation, term, and arithmetic_expression as well as the extracted operator productions agree. Before extracting the additive operators, we first transform the production unary_arithmetic_expression. We factor out the nonterminal term and use the generated epsilon in the inner choice to massage the inner choice into an optional. Next, we inline the production unary_arithmetic_expression and then extract the additive operators. The relational and multiplicative operators can be extracted without prior transformation.

Primary

In the OpenModelica definition of the nonterminal primary, a terminal UNSIGNED_INTEGER exists as in JModelica. Consulting the OpenModelica lexer, we can confirm that the difference lays in the handling by the parser. The regular expressions are identical enabling us to eliminate the choice with the terminal UNSIGNED_INTEGER.

The other choices do not converge, though we will be able to align large parts of the primary definition except the expression involving the keyword initial and the choice including curly braces. We will revisit the choices through the referenced nonterminals in the upcoming paragraphs when we continue our way through the comparator output and its fail notes.

Component References

Reading through the definition of references in OpenModelica, we see that the production component_reference2 has a terminal "OPERATOR", which does not occur in the expressions section of the Modelica specification. Therefore, we will not be able to fully align the references. But we can align the rest of the definition. First, we need to massage the optional expression into a choice with an epsilon. To save us some transformations and writing, we extract the sequence ("IDENT" | "operator") array_subscripts?. The next step is to use the transformation distribute. As the transformation pushes every inner choice outside, the first choice "IDENT" | "OPERATOR" in the definition would also be distributed and we would have to reverse that. After the distribution, we can deyaccify the definition and massage the new definition to have the star expression at the end. If we inline both our helping productions and the production component_reference2, we have aligned the structure of the references. The last step is to look at the lexer definition of the terminals WILD and ALLWILD. The lexer maps the terminal WILD to
an underscore and the terminal ALLWILD to two successive underscores. The definition of the terminal IDENT allows for the same constructs. We can construct the underscore, for example, by not taking the option of a dot and an array subscript and choosing the identifier instead of the operator while not expanding the expression through the iteration. Consequently, we can eliminate the two choices of terminals.

Output Expression Lists

For the output expression lists, we need the right parenthesis to move out of the definition. To reach this goal, we use several transformations. After we transform the definition by distribute, we factor out the right parenthesis of two choices and the sequence "," output_expression_list of the other two. We massage the generated inner choices with an epsilon expression into an optional expression. With the two top-level choices, we can deyaccify the expression, which leads to an expression without any recursion or inner choice and the right parenthesis at the end. If we inline the definition and extract it without the right parenthesis, the alignment is complete.

Matrix Expression Lists

The matrix expression lists need a sequence of familiar transformations: massage to replace the optional, distribute, vertical, and deyaccify to remove the recursion, and massage again to move the star expression to the back of the production. If we inline the transformed production, we have aligned the choice in the production primary that referenced the matrix expression lists. The former matrix expression lists reference the nonterminal expression_list to whose definition we apply the same sequence of transformations as before to align the production.

Function Arguments and For-or-Expression Lists

We will first take a look at the definition of the nonterminal for_or_expression_list2. Transformations massage the optional sequence, distribute the definition, massage the choice of epsilon and the nonterminal expression into an optional, and deyaccify the definition, then inline it.

Next, we consider the definition with the inlined production for_or_expression_list2. The production for_or_expression_list has an epsilon choice, which is massaged into an optional. We will not be able to align the definitions of the nonterminal function_arguments that uses the OpenModelica nonterminal for_or_expression_list. By using familiar transformations and new ones, like unfold, fold, and introduce, we carve out the differences in the productions function_arguments and for_or_expression_list. We introduce the nonterminal function_argument as an alias of the nonterminal expression and fold it into the production function_arguments. We also fold it into the definition of the nonterminal named_argument.

The transformation chain is the following:

```plaintext
introduce(
    function_argument:
    expression
);
fold(function_argument in named_argument);
fold(function_argument in for_or_expression_list);
fold(function_argument in function_arguments);
```

The whole process of transforming the function arguments is rather lengthy and does not add to the convergence of the grammars, only to the localization of the differences. Thus, we will skip a detailed description of the transformations and go on to the function calls.
4.4. MODELICA SPECIFICATION VS. OPENMODELICA

Function Calls

We begin with the production `component_reference__function_call`. The production comprises three branches after transformations which convert the optional into an inner choice and distribute the definition. The branches relate to choices Modelica provides in its definition of the nonterminal `primary`. The OpenModelica nonterminal `function_call` corresponds to the nonterminal `function_call_args` in Modelica. Thus, we rename the nonterminal to its Modelica counterpart. As with JModelica, OpenModelica does not make a difference between names and references. For the sake of further alignment, we introduce the nonterminal `name` with the same definition of component references and replace the call for component references with a call for names. The exact sequence of transformations necessary can be found in Listing E.32 in Appendix E.2. We inline the production `component_reference__function_call` and factor out the nonterminal `function_call_args` to align the structures of the productions `primary`. The difference in the function call arguments lays in the mandatory nonterminal `function_arguments`, which is optional in Modelica. As the whole OpenModelica production `function_arguments` is optional, we can inline the production and extract it without the overall optional. These transformations converge the definitions of the nonterminal `function_call_args`.

Array Subscripts

The last fail note refers to the definition of the nonterminal `array_subscript`. We need the transformation sequence we have used for the production `matrix_expression_list`. The sequence generates the expression that the Modelica specification defines in its production `array_subscript` between the squared brackets. If we inline the production `subscript_list`, the definitions agree.

After we have processed the comparator output, we stop programming transformations. Next, we look at the LCI with the transformations and its output in the next subsection.

4.4.3 Results

With the transformations from the previous subsection, we write the final LCF, see Listing C.7 in Appendix C.4, and run the LCI with it to get the remaining differences. The comparator script outputs the details specified in Listing D.4 in Appendix D.2. Similar to the LCI with JModelica, the Modelica specification and OpenModelica do not converge. The transformed OpenModelica grammar is shown in Listing A.6 in Appendix A.3. Table 4.2 summarizes the result. We will go through the failing and exclusive productions in the following paragraphs.

Expressions, Function Expressions, and Names

The OpenModelica definition of the nonterminal `expression` encompasses all choices the Modelica specification definition includes. Additionally, OpenModelica references the nonterminals `code_expression` and `match_expression` as individual choices. They stay undefined in our limited view of the grammar. Another difference lays in the star expression of an inner choice that comes before the mandatory call to the nonterminal `simple_expression` if choosing the simple expression branch. The simple expressions are referenced in JModelica by the following expression.

\[
( \text{simple_expression} \ "::" | \text{"IDENT" "as" } )^* \text{simple_expression}
\]

By means of this star expression, we produce a sequence of simple expressions. The simple expressions are separated by double colons. Alternatively, an identifier, or a list of such, is assigned to
The subsequent simple expression. This kind of listing and identification is not part of the Modelica specification. The definition of the referenced nonterminal part_eval_function_expression is also part of the Modelica grammar, namely the production function_argument. In Modelica, we have either curly braces around such a function expression or a name before it and parentheses around it. The function expression itself differs in that the names in OpenModelica allow more than the name in the Modelica specification. The production name with only dots and identifiers alternating is much more restrictive than the OpenModelica names which allows subscripts and the keyword operator. Furthermore, the Modelica specification references an optional nonterminal named_arguments where OpenModelica calls for the optional nonterminal function_arguments. OpenModelica contains the same choice as the Modelica specification with the specification's reference to named arguments but increases the possibilities due to the reference to the nonterminal function_arguments. The Modelica specification permits a list of named arguments resulting in a list of the sequence "IDENT" "=" function_argument. Besides the list of named arguments, OpenModelica allows a list of the nonterminal function_argument. At the same time, it is possible to have an expression involving the keyword for or to append named arguments after the list of function arguments. In summary regarding the production expression, OpenModelica is more permissive.

Primary, For-or-Expression List, and Names

Looking at the definitions of the nonterminal primary, we observe that the versions vary in OpenModelica referencing the nonterminal for_or_expression_list instead of the nonterminal function_arguments, which the Modelica specification references. The Modelica specification production body of the function arguments,
function_argument (""," function_arguments | "for" for_indices)? | named_arguments collides with the JModelica definition of for-or-expression lists,

\[
\text{function_argument (""," (function_argument ",")* function_argument? | "for" for_indices)? }
\]

The OpenModelica production for_or_expression_list lacks the reference to the nonterminal named_arguments. The specification generates a possibly empty list of the nonterminal function_argument, separated by commas, and can be extended by either a for expression, in which case we need at least one element in the list, or the nonterminal named_arguments. OpenModelica also produces a list of the nonterminal function_argument though the list is generated slightly different. The list can be empty as well but no named arguments can be appended. Additionally, OpenModelica only allows one function argument before the for expression and not a list that goes beyond one element. The slight difference in the list generation becomes evident in the optional last function argument. The option means the list can end with a comma.

The other difference lays in what the Modelica specification models as a sequence with an inner choice, namely (name | "initial" | "der") function_call_args. OpenModelica generates empty parentheses after the terminal "initial". The choice with the terminal "der" is identical.

The productions name vary in the specification not allowing subscripts, as discussed before.

Named Argument, Component References, and the Keyword operator—

The differences in the two productions named_argument and component_references equal. Instead of an identifier, the inner choice "IDENT" | "operator" exists with the keyword operator as an additional choice.

Function Arguments

OpenModelica also has a production function_arguments though the choice in the definition of the nonterminal primary references the nonterminal for_or_expression_list. The definitions for this production do not converge, either. The core of the OpenModelica definition is the same as the definition of the nonterminal for_or_expression_list without the outer optional. Additionally, the nonterminal named_arguments is a choice and an optional nonterminal at the end of the core of the for_or_expression_list production:

\[
\text{function_argument (""," (function_argument ",")* function_argument? | "for" for_indices)? named_arguments? | named_arguments }
\]

Apart from the difference in the generation of the list of the nonterminal function_argument, the difference boils down to a nonterminal named_arguments allowed after the for expression and the single expression in front of the terminal "for". The list ending on a comma now makes sense as we can have named arguments with a comma as a separator as the last element in the list. But we can also have a named argument after the list without a comma. A difference we carry from the for-or-expression lists is the one element OpenModelica allows before the for expression whereas the Modelica specification models a list with at least one element. In addition, OpenModelica permits named arguments after a for expression which the specification does not generate.

The productions function_argument diverge, which is explainable due to the varying position of the function expression within the grammar. The Modelica specification offers the choices of a function expression or the nonterminal expression, which leads back to the beginning of the grammar excerpt. As OpenModelica already references the function expression at the top nonterminal expression, having the function expression in the definition of the nonterminal function_argument as an alternative is superfluous.
4.4.4 Summary

The two grammar versions have a similar structure in parts of the grammar. Simple renamings have helped aligning a handful of productions. The differences that remain lay in the expressions, the names and component references, named arguments, and for-or-expression lists as well as the definition of the nonterminal primary. We omit the function arguments for a moment. All but the last two have OpenModelica being more permissive. Only with the definition of the nonterminal primary and the production for_or_expression_list referenced in there, the Modelica specification becomes more permissive. The first difference regards the empty parentheses after the keyword initial, the second the missing reference to the nonterminal named_arguments in the production body of the nonterminal for_or_expression_list. A special case is the production function_arguments, which is more permissive in OpenModelica concerning the named arguments allowed after a for expression but more restrictive concerning the number of arguments allowed before a for expression.

After the completion of the comparison of the Modelica specification and OpenModelica, we proceed to the last comparison of JModelica with OpenModelica.

4.5 OpenModelica vs. JModelica

The LCI of JModelica and OpenModelica marks the comparison of the grammar versions of open-source implementations of Modelica. The creators of each version have made independent decisions on design strategies and the implementation of certain features of Modelica. With the previous results of the comparisons of the two implementation versions against the Modelica specification, we can assume that we will not be able to converge JModelica with OpenModelica. We will more likely get a union of the previously reported differences between each version compared to the specification, with one or two exceptions. Fifteen of the 27 productions in the Modelica specification converge in both LCIs. If we compare the previous results, we have a number of production failures that are exclusive to one version of the grammar and some failures present in both LCIs. Exclusive failures mean that the other grammar version converges to the Modelica specification in the respective productions. These differences will still be present comparing OpenModelica and JModelica. The exclusive JModelica differences are the different forms of logical expressions, the number of signs within an arithmetic expression, and the output expression list that is a singleton. The language convergence process of OpenModelica revealed differences in the productions expression and named_argument where JModelica and the Modelica specification converged. As mentioned before, these differences concern the additional choices of match and code expressions, the listing of simple expressions, and a differently placed function expression as well as the keyword operator. They will still be the present in the new LCI. The productions primary, component_reference, function_arguments, and function_argument fail in both LCIs. These productions are the only ones with a chance at convergence. At the end of this last LCI, we will see that none of these productions converges.

As we have restricted the transformations to semantics-preserving ones, the final grammar versions at the end of the previous LCIs have the same expressiveness semantics-wise as the original ones. We will use the transformed versions of OpenModelica and JModelica as the starting points for the new LCI. These versions already have a similar structure due to the transformations towards the Modelica specification and share a large set of common nonterminals. Plus, the definitions are already familiar through the previous sections and more concise than the original ones. As the OpenModelica grammar has resembled the specification grammar in its structure from the start, some transformations would repeat itself to align JModelica’s grammar to the OpenModelica grammar. Thus, the use of the transformed versions will lead to
fewer transformations and a faster result. On a side note, performing the LCI with the original grammar versions has lead to the same results, as expected.

Again, the following subsections go through the initial LCI in Subsection 4.5.1, the transformations (Subsection 4.5.2), and the final LCI in Subsection 4.5.3 ending with a summary of this section’s results (Subsection 4.5.4).

4.5.1 Initial LCI

The initial LCI in this section requires a LCF file with OpenModelica and JModelica as the sources and input branches to the target. As we start with the transformed grammars, we already have the grammars in BGF. Therefore, we use the extractor from the small example in Section 3.55 that uses a script, which simply copies the text of one file into another file. The corresponding LCF file can be found in Appendix C.5. The output tells us that the target fails. The comparator output, as seen in Listing D.5 in Appendix D.3, details the result by specifying that fifteen nonterminals immediately agree. This result does not surprise given the preceding observations about the different comparison results. The differences lay in the expression, arithmetic and logical expression, the primary definition, function and named arguments, names, and component references. We will further align the grammars in the next subsection.

4.5.2 Transformations

If we look at the differences, we observe that we cannot align most of them. The only production that converges after transformation is part_eval_function_expression. We can extract the function expression from one choice of the production function_argument and fold the nonterminal function_call_args into it. Moreover, we can eliminate the nonterminal name which we have kept for reasons intrinsic to the Modelica specification. As it shares the definition with the nonterminal component_reference, we can equate the two definitions and thereby, eliminate the nonterminal name. Because we have introduced the names in both grammars and to not introduce a new difference, we will actually write a transformation for both grammars to eliminate the names. Additionally, we can introduce the nonterminal for_or_expression_list to dissolve this nominal mismatch. If we define it by the nonterminal function_arguments, fold it into the production primary and unfold the definition of function_arguments into the production for_or_expression_list, we have further aligned the production primary and moved the difference to the production for_or_expression_list. Appendix E.3 lists the JModelica transformations within this LCI.

4.5.3 Results

With only a handful of transformations, the JModelica version does not change much (for the LCF, see Listing C.9 in Appendix C.5). The transformed version of JModelica with for-or-expression lists and an own partial evaluation function expression, but without names is in Listing A.4 in Appendix A.2. Listing A.7 in Appendix A.3 shows the OpenModelica grammar without names. The output of the comparator script is shown in Listing D.6 in Appendix D.3. The summary can be found in Table 4.3.

The same productions that failed exclusively in one of the LCIs before are failing within this LCI, as well. For a detailed description of the differences causing these failures, refer to the comparison of the Modelica specification to JModelica and to OpenModelica in the previous sections. The productions that failed in both the previous LCIs still fail. Additionally, we have the failing production for_or_expression_list. We will take a closer look at the underlying differences causing these failures that were present in the two prior LCIs in the following paragraphs.
### Converging Productions vs Failing Productions

<table>
<thead>
<tr>
<th>Converging Productions</th>
<th>Failing Productions</th>
<th>Exclusive OpenModelica Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>array_subscripts</td>
<td>expression</td>
<td>logical_term</td>
</tr>
<tr>
<td>subscript</td>
<td>logical_expression</td>
<td></td>
</tr>
<tr>
<td>annotation</td>
<td>arithmetic_expression</td>
<td></td>
</tr>
<tr>
<td>simple_expression</td>
<td>factor</td>
<td></td>
</tr>
<tr>
<td>relation</td>
<td>primary</td>
<td></td>
</tr>
<tr>
<td>term</td>
<td>output_expression_list</td>
<td></td>
</tr>
<tr>
<td>function_call_args</td>
<td>function_arguments</td>
<td></td>
</tr>
<tr>
<td>named_arguments</td>
<td>function_argument</td>
<td></td>
</tr>
<tr>
<td>expression_list</td>
<td>named_argument</td>
<td>component_reference</td>
</tr>
<tr>
<td>comment</td>
<td>for_or_expression_list</td>
<td></td>
</tr>
<tr>
<td>rel_op</td>
<td>expression</td>
<td></td>
</tr>
<tr>
<td>add_op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mul_op</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string_comment</td>
<td></td>
<td></td>
</tr>
<tr>
<td>logical_factor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>part_eval_function_expression</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3: Final Converging and Failing Productions between OpenModelica and JModelica

### Primary

The differences in the definition of the nonterminal `primary` are basically the same as from the comparison of JModelica and the Modelica specification. They lay in the nonterminal that JModelica references after the terminal "der" and the non-existing choice involving the keyword `initial`. Though, OpenModelica did not converge with the specification in this choice as OpenModelica only had empty parentheses and not a reference to `function_call_args`, we still have the difference that JModelica does not have the keyword `initial` at all. The difference concerning the der expression is the reference to the nonterminal `expression` instead of the nonterminal `function_call_args`.

### Function Arguments

Similar to the previous comparisons, a slight difference in the generation of the list of the nonterminal `function_argument` exists. This difference leads to OpenModelica allowing a named argument after the list of function argument elements without a comma in between and a list ending on a comma. Another difference regards the part with the terminal "for". JModelica allows one nonterminal, namely `expression`. OpenModelica allows a function argument and a named arguments element at the end. The nonterminal `expression`, which is referenced instead of the nonterminal `function_argument`, does not lead to further differences. The function argument in OpenModelica is solely defined by `expression`. In this case, the difference ceases to exist. The difference about the named arguments after the for expression persists.

The difference in the definitions for the nonterminal `function_argument` relates to a difference encountered between OpenModelica and the Modelica specification. JModelica defines the function argument by a general expression or a function expression like the Modelica specification does. As OpenModelica places the function expression as a choice of its production `expression` and only references a general expression in its function argument, the grammar versions do not contradict but express the same at this certain point in the grammar. The actual difference lays in the definition of `expression`, which we already considered.
Component References

The difference between the definitions of component references is a union of the differences we encountered before. First, OpenModelica allows the keyword `operator` which JModelica does not include. Second, OpenModelica allows an array subscript after the first identifier if the reference starts with a dot.

For-or-Expression Lists

The definition of the nonterminals `for_or_expression_list` and `function_arguments` are the same in JModelica. In OpenModelica, the production `for_or_expression_list` misses the references to the named arguments and is fully optional compared to its function arguments. Consequently, apart from the additional named arguments in JModelica, we again have the difference that OpenModelica allows a function argument before the for expression which basically means the same as the expression that JModelica references and which we have discussed before.

4.5.4 Summary

The two grammar versions differ the most in the three comparisons conducted as we cannot converge any common differences they have had with the Modelica specification. We find the slight differences in the logical and arithmetic expressions as well as the greater differences in the expressions. The only points where they agree more are concerning the function expressions and the single element in front of the for expression. In summary, OpenModelica is more permissive in the component references, the definition of primaries, named and function arguments, output expression lists, and expressions itself. The only production where JModelica is more permissive, apart from the small differences in the logical and arithmetic expressions, concern for-or-expression lists where JModelica allows named arguments at all.

The final results of our comparison of OpenModelica and JModelica mark the end of the LCI for Modelica expressions. We will end this chapter with a summary of the work presented and the results we have obtained before we go on to the evaluation.

4.6 Summary of the Language Convergence for Modelica

The LCI delivered a comparison of three grammar versions of Modelica. The question at hand has been whether the three versions generate the same language. To this end, we have attempted to converge the grammar excerpts in pairs. The actions taken towards the answer of the question and the results of the investigation have been described in this chapter. For all three versions, we have written extractors to transform the grammar sources into BGF. We have carried out an initial LCI to learn the differences between the grammars. Then, we have programmed transformations to dissolve the differences and align the grammars. The transformations used are semantics-preserving to preserve the expressiveness of each grammar. At the end, we have conducted another LCI with the transformations specified to find the remaining differences.

In summary, the three versions all differ from each other. We are now interested in what the differences imply if working with Modelica and different simulation environments. With the results of the LCIs at hand, we continue with the evaluation of the work.
Chapter 5

Evaluation

This chapter aims at evaluating the results of the three comparisons conducted in the previous chapter. In this chapter introduction, we will summarize the results to get an overview of the differences and assemble questions arising from these results. The questions guide the evaluation conducted in the following sections.

The summary is given in tables where each row concerns one construct in the grammar and the columns represent the two grammar versions in comparison. Gray-colored cells highlight the more relaxed version. An empty cell implies the feature described in the adjacent cell being not present in this grammar construct.

Table 5.1 summarizes the results of the comparison of the Modelica specification and JModelica. As one can see, the grammar versions are more permissive in some cases and more restrictive in others. In four constructs, the Modelica specification is more permissive, namely, primaries, component references, function arguments, and output expression lists. In three cases, i.e., logical expressions, names, and single function arguments, JModelica is more relaxed. Concerning arithmetic expressions, both grammars contain features that extend the other grammar’s expressiveness.

Table 5.2 presents the differences between the Modelica specification and OpenModelica. OpenModelica extends the expressiveness of the Modelica specification in terms of expressions, simple expressions, names, component references, and named arguments. The first two differences manifest themselves in the production expression. The last three differences concern the additional keyword operator and the fact that OpenModelica does not distinguish names from component references. The Modelica specification models the for-or-expression lists like function arguments, allowing more than OpenModelica. The specification also permits input arguments for the function initial(). Function arguments are split in their expressiveness. On the one hand, function arguments are more relaxed in OpenModelica with the possibility of named arguments after a for expression. On the other hand, function arguments are more permissive in the specification with a possible list of function arguments before said for expression.

Table 5.3 displays the results of the LCI for OpenModelica and JModelica. OpenModelica has a higher expressiveness in seven cases. The arithmetic expression, as between the Modelica specification and JModelica, are more relaxed concerning the allowed operators as signs, but more restricted concerning the position and the number of the signs. The cases of JModelica having a higher expressiveness than OpenModelica include for-or-expression lists as JModelica, like the Modelica specification, does not restrict its function arguments to the slimmer version of for-or-expression lists as OpenModelica does. The other case is the order of connectors in logical expressions.
### Modelica Specification | JModelica
---|---
**Logical expressions** | DNF | Arbitrary order
**Arithmetic expressions** | Optional sign in the beginning of an arithmetic expression | Multiple signs in front of power expressions
| Permitted operators as signs: | Permitted operators as signs: |`````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````````
In all three comparison, we observe that no distinct relation exists between any of the grammars. From this result, new questions arise, including:

- What does one grammar generate that the other grammar does not?
- Can we generate something similar to what the other grammar can produce?
- What might be the origins of the difference?
- On a broader scale, is one grammar version a super-/subset of another?
- What do we have to bear in mind switching from one simulation environment to the other?

We will focus on these questions in the first section of this chapter, Section 5.1. As a grammar engineer, other questions emerge:

- What transformations have been used and what transformations have not been used?
- Would different transformations lead to different results?
- Does the order of application matter?
- In conclusion: Is the outcome of the comparison correct?

Section 5.2 delves into a discussion of these questions. At the end of this chapter, we consider the LCI as the means to answer our research question (Section 5.3).

### 5.1 Analysis of the Results

In this section, we attempt to evaluate the results collected during the comparison of the Modelica grammar versions. We start with a brief inspection of the parse trees before moving on to the implications of the differences on the generated language (Subsection 5.1.1), the overall relation (Subsection 5.1.2), and the implications on the use of different simulation environments (Subsection 5.1.3).

The comparison of the different Modelica grammars has yielded transformed grammars for JModelica and OpenModelica. These transformed versions resemble the grammar structure of the Modelica specification. The parse trees of the transformed grammars echo the semblance; see Figures 5.1 and 5.2. Figure 5.3 replicates the original Modelica specification tree, shown in Figure 2.1 for the sake of convenience. We have conducted a quick comparison of the original parse trees in Section 2.1. If we now compare the original Modelica specification parse tree with the parse trees of the transformed grammars, we observe an alignment of the tree structures in comparison to the previous tree structures. The OpenModelica parse tree still differs from the other parse trees by the three additional nodes referenced by the node `expression`. Additionally, the lower part of the tree appears to be more complex. This impression is due to the exclusive node `for_or_expression_list` and one of the nodes referenced by the node `expression`, namely `part_eval_function_expression`. The tree visualizes that differences, apart from variations in the definitions, still exist, especially in the modeling of function arguments. The JModelica tree reflects the structure of the specification tree more closely than OpenModelica. The varying parts of the trees concerning function arguments highlight a major point for differences between the grammar versions. In a converging grammars scenario, the parse trees would have been identical after transformation. But contrasting the transformed parse trees against the original specification parse tree reflects the inherent differences between the various grammar productions.

In the next subsection, we will focus on the implications of the differences between the grammars. The first three questions in the list at the beginning of this chapter guide our evaluation of the results of the LCIs.
Figure 5.1: Parse Tree of the Transformed JModelica Expressions
Figure 5.2: Parse Tree of the Transformed OpenModelica Expressions
Figure 5.3: Parse Tree of the Modelica Specification Expressions
5.1. ANALYSIS OF THE RESULTS

5.1.1 Implications of the Individual Differences

In this subsection, we will analyze what one grammar can generate that the other grammar does not include and how close we can come to remodel that rejected expression in the other grammar. Moreover, we will try to find a reason for the occurrence of the difference. Before starting with the analysis, a short note on grammars from specifications and from implementations. Implementation grammars, such as the ones extracted from the OpenModelica and JModelica parser descriptions, are usually more permissive than grammars from language specifications [11]. OpenModelica and JModelica work on the language Modelica. Hence, whenever their grammar versions are more permissive compared to the Modelica specification, the consequences are acceptable as the grammars still accept the specification’s version.

Modelica Specification vs. JModelica

The constructs considered within this paragraph are logical and arithmetic expressions, primaries, output expression lists, function arguments, component references, and names.

Logical Expressions The first difference between the Modelica specification and JModelica appears in the productions for logical expressions. The specification only allows logical expressions in DNF. JModelica does not restrict the logical expressions to this normal form. JModelica allows an arbitrary order of conjunctions and disjunctions, which the Modelica language does not permit. The Modelica language specification document details that a disjunction has a higher precedence than a conjunction. The higher precedence leads to a logical expression automatically being in DNF if no parentheses are given. The grammar reflects this circumstance by using two productions to model the logical expressions. JModelica does not define any precedence between the two logical connectors. Without any parentheses, a logical expression is parsed from left to right regardless of the connectors. To determine another order of parsing, parentheses are necessary. The grammar reflects this behavior. As every logical expression of disjunctions and conjunctions can be transformed into DNF, the logical expressions have the same expressiveness. But concerning the expressiveness of the grammar, JModelica is more permissive. A reason behind JModelica choosing not to restrict the logical expressions might be in that normalizing logical expressions potentially leads to much larger expressions with a length up to the exponential of the original. Therefore, JModelica has possibly decided for convenience reasons to not restrict the expressions to DNF.

Arithmetic Expressions Arithmetic expressions vary in the characteristics of a sign. The Modelica specification allows an optional sign at the beginning of an arithmetic expression and only additive operators as the indicator for the type of the addition afterwards. Each term within the addition can have numerous multiplications of factors that can be a power expression or a simple primary. These factors are all implicitly positive. In the arithmetic expressions of JModelica, each factor of a multiplication can have an arbitrary number of algebraic signs ahead of a power expression or a simple primary. This production implicates that the overall arithmetic expression can have a leading sign. But the position of the sign is not restricted to the forefront. Considering the workings of mathematics, numerous signs can be collapsed into a single one with the common rules of two minuses becoming a plus and a minus and a plus becoming a minus. The result of a multiplication is either positive or negative. As a consequence, a multiplication of signed numbers can be expressed as a multiplication of unsigned numbers with a corresponding sign in front of the term that consists of the multiplication. A single sign in front of a term can be merged with the additive operator between the terms that leads to an arithmetic expression with terms separated by additive operators and an optional sign at the beginning, the absence of a sign likewise
indicating that the term is positive. Thus, the expressiveness concerning arithmetic expressions is identical as we can transform one arithmetic expression modeled according to JModelica into an arithmetic expression modeled according to the Modelica specification. The difference, again, lies in the expressiveness of the grammar and not in the mathematical expressiveness of the arithmetic expressions, with JModelica being more permissive. Another difference is that JModelica only allows a plus or minus as a sign whereas the Modelica specification permits all its additive operators, which include bitwise addition and subtraction.

**Primaries** In the examination of the arithmetic expressions, we have come across primaries, which we deal with next. The collisions in this production are between the following two excerpts of the respective Modelica specification (the first excerpt) and JModelica (the latter one) production.

\[
("\text{der}" \mid "\text{initial}") \text{ function_call_args} \leftrightarrow "\text{der}" \text{ "(" expression ")"}.
\]

The keyword initial references the built-in function initial() in Modelica. Within JModelica, the keyword initial does not appear in this context. The JModelica user’s guide reveals that the function initial() is only partially supported. JModelica allows different kinds of models to be created and translated through different compile functions. Not all models include this built-in function. Hence, the reason this particular keyword is not part of the grammar may lie in the inconsistency of handling the function initial(). The nonterminal following the keyword der exhibits the next and last difference. The referenced nonterminal function_call_args allows an optional nonterminal function_arguments between parentheses. JModelica demands an expression. Going through the grammar again, starting from the referenced nonterminal expression, makes it possible to put almost the same expression after the keyword der. But to actually reach the production function_arguments in the next round, the production primary either adds curly braces or a nonterminal name and opening and closing parentheses. The JModelica representation is, therefore, more restrictive.

**Output Expression Lists** Output expression lists represent a similar problem. JModelica allows one expression, whereas the Modelica specification allows a possibly empty list of expressions separated by commas. In JModelica, similar lists are possible with a trick: Starting from the nonterminal expression, we choose the sequence name function_call_args, adding the nonterminal name, or the sequence {" function_arguments "}, in the production primary. The nonterminal function_arguments models a sequence function_argument ("," function_argument)∗ where we can replace each nonterminal function_argument with its first choice expression. By choosing the sequence name function_call_args, we can model an empty list. At this point, it is worth mentioning that the Modelica specification allows a list of commas in parentheses. It is debatable how much sense a list of commas makes and whether the compiler accepts it.

**Function Arguments** The aforementioned function arguments differ in one point. The difference is the one expression that is allowed before the terminal for in contrast to a list of the nonterminal function_argument separated by commas. As before, we could go through the grammar again and emulate the same part in front of the terminal for with an additional name or curly braces around the part. Taking into consideration previous Modelica specifications, we discover that before the most recent version, no production function_argument exists. The production function_arguments referenced the nonterminal expression. This development partly explains why JModelica references an expression before the terminal "for" but it does not give hints as to why JModelica does not allow a list of expressions and why it has changed.
5.1. ANALYSIS OF THE RESULTS

the expressions reference to a function argument reference for the rest of the production but the for expression. The definitions of the nonterminal function_argument collide in the allowed function arguments. JModelica references the nonterminal function_arguments instead of the nonterminal named_arguments. That makes JModelica more permissive.

Component References and Names The last difference in the result of the Modelica Specification vs. JModelica comparison refers to component references and names. As mentioned before, the specification restricts names to alternating dots and identifiers while JModelica models the names in the same way as component references. Component references allow array subscripts after identifiers. The Modelica specification uses the production name for specifying function names and paths. Function names and paths do not have array constructs in their name; to incorporate subscripts in the grammar production is inaccurate. Thus, the specification models the names and component references more precisely, only permitting subscripts where they are justified. The difference between the definitions of the component references boils down to only allowing subscripts after two times a dot and an identifier if the reference starts with a dot. The origins for this difference are unclear. The evolution of the language Modelica has lead to the recent introduction of a dot at the beginning of a component reference regarding the grammar. This initial dot is incorporated by JModelica but not to the same extent.

We discuss the differences encountered between the Modelica specification and OpenModelica in the next paragraph.

Modelica Specification vs. OpenModelica

The productions expression including its referenced nonterminals and their definitions if available, primary, for-or-expression-list, named_argument, and component_reference are the topics of this paragraph.

Expressions We start with the differences in the definition of the nonterminal expression. All differences are instances of OpenModelica being more permissive. The two big differences, we cannot explain with our restriction of the grammar, are the references to the nonterminals code_expression and match_expression. The nonterminals model constructs that are part of the Modelica specification, as well. The reason behind placing them at this point in the grammar is not obvious, equally the question whether the definitions agree. This problem must be shifted to future work. As already discussed in the previous chapter, we can build a list of simple expressions separated by double colons or the sequence "IDENT" "as". The Modelica specification does not include a list of simple expressions with these types of separation. The definition of the referenced nonterminal part_eval_function_expression can be found as a part of the Modelica specification production function_argument. To reach the function expression in the specification, the grammar at least generates either curly braces or a name and parentheses.

Function Expressions and Names The differences between the function expressions themselves lay in the definition of the name in OpenModelica versus the name definition in the Modelica specification and using the nonterminal function_arguments instead of the nonterminal named_arguments. The production name with only dots and identifiers alternating is much more restrictive than the OpenModelica name, which is equal to component references and as such allows subscripts and the keyword operator. A reason behind the restrictiveness of the Modelica specification concerning names has been discussed in the previous paragraph. The differing arguments in the function expressions mean that OpenModelica allows more than the specification. The specification permits a list of named arguments resulting into a list of the sequence
"IDENT" "=" function_argument separated by commas. Additionally, OpenModelica allows a list of the nonterminal function_argument. At the same time, it is possible to have an expression involving the keyword for or to append named arguments after the list of function arguments. The origin of the difference is unclear. The production function_arguments exists in OpenModelica, as well. The difference lies in the number of expressions allowed in front of the terminal "for" and to append a list of named arguments after such a for expression. The succeeding named arguments make OpenModelica more permissive. The list of function arguments preceding the for expression is discussed in the next paragraph.

Primaries and For-or-expression lists The next differences we examine are in the definition of primaries and the referenced construct for-or-expression lists. Primaries in the Modelica specification include the keyword initial followed by the nonterminal function_call_args. In OpenModelica, the keyword is followed by empty parentheses. Referring to the language documentation for Modelica yields that the built-in function initial() has no input arguments. This information validates OpenModelica's version with explicitly allowing only empty parentheses. The Modelica specification presents a grammar that permits arguments for the function initial(). As a side note, the primaries also include the keyword der followed by the non-terminal function_call_args. Both versions allow empty parentheses after the keyword der. The keyword plus the function call arguments model the function der(). From the language document, we learn that the function, which returns the derivative of the input, needs to have input parameters. Thus, the grammars do not strictly reflect the requirements for this function. In the last difference between the primaries, OpenModelica references the nonterminal for_or_expression_list whereas Modelica calls for the nonterminal function_arguments in the choice, which includes curly braces. As described during the comparison of OpenModelica and the Modelica specification, the for-or-expression list lacks a reference to named arguments and allows only one element before the terminal for. We could build a list in front of the “for ” part by wandering through the productions again and would end up with an additional name and parentheses or curly braces. The named arguments can be reached if we designate the last function argument in the list to model the named arguments and walk through the grammar again. Until we arrive at named arguments, a name and parentheses have been added. A minor difference in generating the list of function arguments exists, which allows the list in OpenModelica to end on a comma. It is arguable how reasonable this way of modeling the list with a possible ending comma is. The comma becomes valid when modeling the nonterminal function_arguments in OpenModelica. In its definition, a named argument can end the list, which warrants a comma as a separator. With this comma, OpenModelica follows the Modelica specification in separating elements of a list by commas. But the production also allows one to not place a comma between the last function argument and the ending named arguments, which does not follow the specification's approach of element separation in lists.

Keyword operator The last difference investigated in this paragraph regards the keyword operator, which OpenModelica uses in its definition of the nonterminals named_argument and component_reference. The remainders of the definitions agree. Within the Modelica specification, the keyword operator only exists as a class prefix in the production class_prefixes. The exact motive behind embedding the keyword in these productions is not visible but may refer to operator overloading. We will take a closer look at the keyword and its possible meaning in the chapter about the implications of the differences on using different simulation environments.

The implications of the differences between OpenModelica and JModelica follow in the next paragraph.
5.1. ANALYSIS OF THE RESULTS

OpenModelica vs. JModelica

The differences between OpenModelica and JModelica form two classes. One class contains the productions where one grammar converges with Modelica while the other does not. The implications of these differences are the same as mentioned in the previous two paragraphs. For example, the difference in logical expressions between OpenModelica and JModelica is the same as the difference between JModelica and the Modelica specification. OpenModelica restricts logical expressions to DNF like the Modelica specification does. The difference continues to exist with the comparison of OpenModelica and JModelica. The differences that belong to the second class stem from both grammars failing to agree with the Modelica specification. As by the result of the LCI of OpenModelica and JModelica, these differences do not collapse but persist, the exception being function expressions. In this comparison, the differences are basically a union of the differences encountered before. The component references, for example, diverge between OpenModelica and the Modelica specification because of the additional keyword `operator`. The LCI of the Modelica specification and JModelica has revealed that JModelica does not allow a subscript after the first identifier if the reference starts with a dot. The differences merge when comparing OpenModelica and JModelica. OpenModelica includes the keyword `operator` and allows said subscript.

After analyzing the individual differences, we investigate the general correspondence between the grammars. We focus on the question whether one grammar is a sub- or superset of another.

5.1.2 Overall Relation and Equivalence

With the differences still remaining between the grammars after transformation, we can state that the grammar versions do not correspond. The grammars are not equivalent, which means, strictly speaking, they do not model the same language. The follow-up question is whether one grammar has a higher expressiveness. In any case, we cannot detect a distinct relation between the grammar pairings. In some cases, one grammar is more permissive, in other cases, it is the other way around, which means they are neither a subset nor a superset of one another. Albeit, we can make some observations concerning the relations and expressiveness of the versions.

The Modelica specification is more restrictive when it comes to names, which both OpenModelica and JModelica do not distinguish from component references. As discussed before, this restrictiveness might reflect a more correct way of expressing names. JModelica is more permissive concerning logical and arithmetic expressions in comparison to the Modelica specification and OpenModelica. Furthermore, the productions `function_argument` in relation to Modelica and `for_or_expression_list` regarding OpenModelica are more relaxed. In all other productions, JModelica is equally or less expressive than the other two versions. The Modelica specification models the component references, function arguments, output expression lists, and the primaries by the input arguments for the function `der()` and by the function `initial()` present in a more permissive way. Overall, JModelica is more restrictive than the Modelica specification apart from the logical and arithmetic expressions and a single function argument. The same holds for the relation of JModelica towards OpenModelica, except logical and arithmetic expressions and for-or-expression lists. OpenModelica has a higher expressiveness with respect to component references, function and named arguments, output expression lists, expressions, and primaries in comparison with JModelica. The only expression, apart from the logical and arithmetic expressions, that is more permissive in JModelica is the for-or-expression lists. Comparing OpenModelica with the Modelica specification yields a similar observation. In addition to the aforementioned definition of the nonterminal `name`, the OpenModelica productions `component_reference`, `named_arguments`, `function_arguments`, except the terms allowed before a for expression, and `expression` are more permissive. The Modelica specification only
exceeds OpenModelica’s expressiveness when it comes to for-or-expression lists and the non-terminal between the parentheses after the keyword initial. Concerning the keyword initial, we have recorded that the possible input for the function initial() in the Modelica specification is erroneous according to the language specification document. In general, OpenModelica is more permissive with two exceptions and the special case of function arguments with being more restrictive and permissive at the same time.

The differences encountered are all cases of one grammar being more relaxed than the other. Exhibiting a difference, one grammar always includes the possibilities of the other grammar but extends them by further references. No definite contradictions exist. In case a code example adheres to a more restrictive production, the more relaxed version will also accept it. One has to be careful with code that complies with the more permissive production where a more restrictive grammar might reject the code.

Ignoring minor differences, the three versions form a loose relation of OpenModelica being a superset of the Modelica specification being a superset of JModelica. With OpenModelica being an implementation of a grammar that is optimized for readability, the first result does not astonish. JModelica, though being more relaxed in some cases, does not express more than Modelica, which is unusual for a grammar derived from an implementation. OpenModelica strives to develop a full implementation of the Modelica language openly available. JModelica does not set its goal to accomplish a complete implementation. The differences in the grammars reflect the varying goals of OpenModelica and JModelica.

The next subsection discusses the implications of the differences on the use of different simulation environments.

5.1.3 Implications of the Result on Using Different Simulation Environments

So far, we have discussed the individual differences and what they mean in terms of words generated by one grammar, but not by the other one. As these grammars describe a programming language, the next step is to evaluate what these discrepancies mean if different simulation environments are fed the same Modelica code. Various scenarios exist. First, the Modelica code adheres to the Modelica specification. If using a simulation environment that uses a more permissive grammar, no problems occur that can be attributed to a differing grammar. If the grammar of the simulation environment is more restrictive, one must be careful as the code might include passages that the more restrictive grammar does not generate. In a second scenario, the Modelica code follows the grammar rules of a simulation environment that has a grammar varying from the Modelica grammar. The programmer has to pay attention if switching to another simulation environment. The code may be rejected by the new environment if the new grammar varies from the other grammar in such a way that it is more restrictive. Writing code for a new simulation environment, the programmer needs to familiarize her- or himself with what is possible within the other grammar and not rely on what the former simulation environment accepts. For the simulation environments OpenModelica and JModelica, the previous subsection has revealed that they do not have a distinct relationship. With a distinct relation between the grammars, we would have been able to state that switching from the more restrictive one to the more permissive one does not lead to differences concerning expressions grammar-wise. The programmer would only need to revise his code if switching from the simulation environment with the more permissive grammar to the environment with the less permissive grammar. Unfortunately, OpenModelica and JModelica do not fully adhere to the Modelica specification and both are more restrictive in some passages and more permissive in other ones, compared to the specification and compared to the other one. Therefore, the programmer has to review his code whenever switching the simulation environments. In the remainder of this subsection, we will give hints on which
expressions might cause problems and where pitfalls lurk.

Expressions and Function Expressions

JModelica adheres to the Modelica specification. Difficulties might arise if presenting JModelica with code written in accordance with the OpenModelica grammar. OpenModelica references match and code expressions in the definition of the nonterminal expression. What the consequences of modeling the expressions this way are in its entirety, we cannot predict. For this excerpt of the grammar, the existence of code and match expressions within the definition of the nonterminal expression means that JModelica rejects code that uses a code or match expression whenever the situation warrants an expression. As mentioned above, a further investigation must be postponed. Another difference of OpenModelica compared to the Modelica specification and JModelica is the reference to a function expression at this stage in the grammar. The implication is that the grammar of OpenModelica models the following minimal expression, a function with the name “someFunction” and no input arguments:

function someFunction()

The JModelica grammar and the Modelica specification do not accept this expression. A corresponding minimal function expression for the specification and JModelica require curly braces, or a name accompanied by parentheses, and an identifier.

{ someId = function someFunction() }
someOtherFunction( someId = function someFunction() )

The OpenModelica grammar accepts the second example, as well. The function expressions allow the function to have input arguments. The Modelica specification requires named arguments, which means that all arguments are of the form "IDENT" "=" function_argument, e.g., \((x = 1, y = 2)\). OpenModelica and JModelica allow function arguments, which results in arguments of the form function_argument possibly followed by named arguments, e.g., \((1, 2, x = 1)\). They also allow a for expression as input argument, which may look like the following expression:

function sum( i for i in 1:10 )

As both simulation environments permit such an expression, the expression is not rejected according to the grammar if changing the environment. The differences in the definition of function arguments are discussed below.

Logical Expressions

Within the logical expressions, two pitfalls exist, one being different evaluation orders, the other being expressions not accepted by the Modelica specification. JModelica reads the following logical expression from left to right with first evaluating the disjunction and then the conjunction coming to the result that the expression is false.

true or x and false

The Modelica specification intends to first evaluate the conjunction followed by the evaluation of the disjunction which leads to the expression evaluating to true. The two boolean values are placeholders for any logical expression that evaluates to the boolean value given a specific assignment of the involved variables in the logical expression. The same expression with parentheses, as depicted below, is possible within JModelica but not within the Modelica specification, as the expression is not in DNF.

(true or x) and false
As OpenModelica follows the Modelica specification in modeling logical expressions, one has to be careful when switching platforms. Feeding JModelica with Modelica code that adheres to the Modelica specification might result into an evaluation of a logical expression not as intended due to the absent precedence in JModelica. The same problem might exist when switching from JModelica to another simulation environment. The evaluation order might change again with precedence present. In addition to the varying evaluation orders, logical expressions allowed in JModelica can be rejected by OpenModelica or other platforms that model logical expressions only in DNF.

**Arithmetic Expressions**

Though the arithmetic expressions have the same expressiveness considering the mathematics behind them, the grammars vary. JModelica accepts the following arithmetic expression whereas the Modelica specification and the OpenModelica parser do not permit such an expression.

\[-2 + (-1)\]

The sign in front of the second summand is not possible in the Modelica and OpenModelica grammars. A more complex expression is:

\[-((-(2))) + (5 \times (-7) \times ((+(+9)))\]

This expression exhibits the ability to have more than one sign before a factor in JModelica. If a bitwise operator is used as a sign in OpenModelica, JModelica rejects the expression according to its grammar.

**Primaries**

The differences in the definition of primaries concern the keywords `initial` and `der` and the list between curly braces. JModelica does not include the keyword `initial`. Without the keyword being a reserved word, it is possible to use it as an identifier. But, as mentioned above, the function that is referenced by this keyword is only partially supported by JModelica. One has to be careful with the function `initial()` if using JModelica. Researching the built-in function `initial()` in JModelica has revealed that several other built-in Modelica functions are not supported within JModelica [6].

The keyword `der` refers to the derivative function. In OpenModelica and the Modelica specification, function call arguments follow the respective terminal "der", which leads to parentheses and a list of function arguments. The grammar generates the expression `der(x,y)`. JModelica references an expression that generates the expression `der(x)`, for example, but not a list of function arguments. The grammar does not allow for a list of arguments in JModelica. According to the grammar, more complex expressions are possible but they are also permitted in OpenModelica and the Modelica specification.

The last difference mentioned in the introductory sentence regards the list generated between curly braces. This failure in convergence involves the productions `function_arguments` and `for_or_expression_list`. We will take a look at them together in the next but one paragraph.

We move on to a nonterminal that represents a choice within the production `primary` surrounded by parentheses, the output expression lists.

**Output Expression Lists**

If a function returns more than one output expression, a list of output variables is needed to use all return values. Output expression lists model this list to whose elements the result expressions are assigned to. JModelica references only one expression, which allows for the
expression \((x)\). OpenModelica and the Modelica specification permit a list of output variables, e.g., \((x, y, z)\). The difference resembles the one we described in the previous paragraph about the input arguments allowed for the function \(\text{der}(\ldots)\). Therefore, code adhering to the Modelica specification and working in OpenModelica should be revised with respect to output expressions if using JModelica.

**Function Arguments and For-or-expression Lists**

We have already taken a look at what function arguments can generate when inspecting function expressions at the beginning of this subsection. The definitions for function arguments all differ between the three grammars. We have already discussed the difference in the list generation of function argument elements. The remainder of the definitions of function arguments in OpenModelica and JModelica comes to an agreement. The difference in list generation leads to JModelica modeling the following expressions:

\[
\text{arg0}, \ \text{arg1}, \ \text{arg2} \\
\text{arg0}, \ \text{arg1}, \ x = \text{arg2}
\]

OpenModelica models the subsequent expressions:

\[
\text{arg0}, \ \text{arg1}, \ \text{arg2} \\
\text{arg0}, \ \text{arg1}, \ \text{arg2}, \\
\text{arg0}, \ \text{arg1} \ x = \text{arg2} \\
\text{arg0}, \ \text{arg1}, \ x = \text{arg2}
\]

The way JModelica models the list generation is in accordance with the Modelica grammar. The second and third OpenModelica example are rejected by the Modelica specification and JModelica. It is arguable even within OpenModelica whether a list is allowed to end on a comma.

The next difference exists concerning the expressions involving the keyword `for`. JModelica and OpenModelica agree on the definition as the referenced function argument before the keyword `for` in OpenModelica is defined solely by an expression. JModelica demands an expression at this position in its definition. The Modelica specification approves a list of function arguments that exceeds the singleton expression of OpenModelica and JModelica. Thus, a code excerpt that the Modelica specification accepts is rejected by the other two grammars. An expression allowed within the Modelica specification but not within JModelica and OpenModelica is:

\[
\text{arg0}, \ \text{arg1}, \ i \ \text{for} \ i \ \text{in} \ 1:10
\]

The first two arguments would not be generated by the JModelica and OpenModelica grammars. On the contrary, OpenModelica permits a list of named arguments after a for expression which both JModelica and the Modelica specification do not model. An example of such an expression is:

\[
i \ \text{for} \ i \ \text{in} \ 1:10, \ x = \text{arg0}, \ y = \text{arg2}
\]

As the title of this paragraph suggests, for-or-expression lists and function arguments relate somehow. These two nonterminals are one cause for the failure of convergence of the definitions of primaries. For-or-expression lists model similar expressions to function arguments. The production name originates from the OpenModelica grammar. OpenModelica does not allow the full spectrum of function arguments between curly braces in the primary definition but only a portion of it. The core definition is the same but lacks the reference to named terminals. As a consequence, OpenModelica does not allow named arguments between curly braces. In addition, it allows empty braces. Expressions generated by OpenModelica include:
Again, the lists are generated with possibly a comma as the end symbol. Lists OpenModelica does not accept include:

\{x = \text{arg}0, \ y = \text{arg}1\}

The other expressions OpenModelica does not accept but the Modelica specification allows concern the list of function argument elements before a for expression, which is discussed before. JModelica and the Modelica specification allow a list of function arguments possibly expanded by a list of named arguments or a list of named arguments as the last example suggests. These differences arise from the additional possibilities that the function arguments offer.

The last constructs we interpret regarding the use of different simulation environments are component references, names, and named argument expressions.

Named Argument and Component References

Component references also model names in OpenModelica and JModelica. Names are more restricted than component references as they do not allow subscripts. The difference has already been discussed above. Writing code adhering to the names scheme will not cause problems within OpenModelica or JModelica. The differences that might cause problems are in the additional keyword \texttt{operator} in OpenModelica and the missing optional subscript after the first identifier if the reference starts with a dot. Examples of component references OpenModelica accepts but JModelica does not are:

\texttt{.x[3:6] = overload(name list)}

The keyword \texttt{operator} is used to model overloading of operators to which the Modelica Association dedicates an own section in its language specification document since Version 3.1. For the operator keyword, any operator, of relational, additive, or multiplicative nature, can be inserted and overloaded as a result. Overloading functions is not possible within JModelica \cite{JModelica}. Overloading is supported by the Modelica specification but is not modeled within its grammar in the same way as OpenModelica does it.

The differences of the productions \texttt{named_argument} and \texttt{component_reference} resemble each other as the definition of the nonterminal \texttt{named_argument} also involves the keyword \texttt{operator} in OpenModelica but not in the Modelica specification and JModelica.

The considerations of the differences caused by the keyword \texttt{operator} close this section of analyzing the results of the LCIs. We have taken a look at the meaning of the differences for the grammars, the overall relation, and the use of simulation environments with different grammars. The next section discusses the transformations used in the three LCIs conducted.

5.2 Analysis of the Transformations

This section is subdivided into the summary of the programmed transformations (Subsection 5.2.1), an investigation of the unused operators (Subsection 5.2.2), and an analysis of the order of the transformations and the correctness of the result (Subsection 5.2.3).
5.2. Analysis of the Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Modelica Specification vs. JModelica</th>
<th>Modelica Specification vs. OpenModelica</th>
<th>OpenModelica vs. JModelica</th>
<th>Overall Number of Transformations</th>
</tr>
</thead>
<tbody>
<tr>
<td>renameN(f, t)</td>
<td>10</td>
<td>2</td>
<td>-</td>
<td>12</td>
</tr>
<tr>
<td>renameT(f, t)</td>
<td>42</td>
<td>46</td>
<td>-</td>
<td>88</td>
</tr>
<tr>
<td>vertical(n)</td>
<td>13</td>
<td>11</td>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>deyaccify(n)</td>
<td>12</td>
<td>10</td>
<td>8</td>
<td>20</td>
</tr>
<tr>
<td>inline(n)</td>
<td>20</td>
<td>16</td>
<td>14</td>
<td>36</td>
</tr>
<tr>
<td>factor(e, e)</td>
<td>17</td>
<td>14</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>massage(e, e)</td>
<td>11</td>
<td>7</td>
<td>22</td>
<td>33</td>
</tr>
<tr>
<td>extract(p)</td>
<td>11</td>
<td>7</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>distribute(n)</td>
<td>1</td>
<td>10</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>unfold(n)</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>yaccify(p+)</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>horizontal(n)</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>removeV(p)</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>introduce(p)</td>
<td>-</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>fold(n)</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>equate(n, n)</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

| Overall Number         | 143                                  | 122                                    | 144                       | 122                             |
| o                      | 7                                    | 2                                      | 2                         | 294                             |
| m                      | 2                                    | 2                                      | 2                         | 246                             |

Table 5.4: Overview of the Transformations Used in the three LCIs
- **o**: overall number of transformations used in the respective LCI
- **m**: number of transformations leading to the resolution of a nominal or structural mismatch

5.2.1 Programmed Transformations

The convergence process has spawned the programming of close to 300 transformations. Table 5.4 lists the concrete numbers on which transformations have been used how many times. As mentioned before, some transformations are programmed to carve out the difference or to shorten the length of an expression during transformation programming by extracting an untouched part of a production and inlining it later. The overall number of transformations needed for the convergence of JModelica and OpenModelica with the Modelica specification is approximately the same, though, OpenModelica resembles the Modelica specification more in the general structure. This overall number consists of all transformations applied to the grammars as listed in Appendix E. The left column of each grammar column in the table, marked by the letter ‘o’ for overall, names the overall number of transformations. The number of transformations that aligned the grammars is 122 for the first two comparisons. For this number, we have taken out the transformations that have served to clarify differences, like the factorizing of the logical factors in the logical expressions of JModelica, or to simplify transformations by extracting part of a production. In other words, this number comprises the transformations that align the grammar sources by either dissolving an exclusive nonterminal or aligning production bodies of shared nonterminals. The right column, headed with the letter ‘m’ for mismatch, gives the number of transformations that have lead to dissolving a mismatch. The transformations mirror the different structures and ways of modeling a construct. OpenModelica has required a lot more massaging than JModelica whereas JModelica has needed almost thrice as much factoring.
OpenModelica has shared two third of its defined nonterminals with the Modelica specification, resulting into fewer renamings of nonterminals compared to JModelica. JModelica has only one third of its nonterminal symbols in accordance with the Modelica specification, which requires more renamings. The comparison of JModelica and OpenModelica is based on the transformed grammars of the previous comparisons. These grammars have each had applied over 140 transformations themselves before the comparison against each other starts. As discussed in Section 4.5, most of the differences both grammars exhibit at this point towards the Modelica specification are bound to persist in this new LCI. The only transformation chain dissolving a nominal mismatch concerns function expressions. The other transformations reverse the introduction of names in both grammars for this comparison as the notion of names has only been introduced to accommodate the Modelica specification.

5.2.2 Unused Operators

As mentioned before, XBGF includes 49 transformation operators. The transformations can be categorized in the following way; the number in brackets denotes how many operators we have used out of this category:

- semantics-preserving operators: 21 (13)
- semantics-changing operators: 18 (1)
- decorative operators: 4 (0)
- rename operators: 4 (2)
- dump (not used)
- reroot (not used)

We have used 16 of the 49 operators. As we have limited ourselves to semantics-preserving alterations of the grammar, we have mainly used semantics-preserving operators. The one semantics-changing operator we have used is removeV to eliminate choices in a production that do not carry any new information and as such, are redundant. The decorative operators account for features of the BGF itself, labels and selectors. As the Modelica grammar versions do not include labels and selectors, we did not need to introduce or eliminate any labels and selectors. The labels and selectors are also the reason why we have only used two out of four renaming operators. Labels and selectors can be renamed as well but without any present, the operators are unused. The grammar versions do not include specific roots, either, making a rerooting of any grammar version unnecessary. The operator dump halts the application of a transformation chain and dumps the current state of the grammar into a log file. As such, it can be used to debug XBGF files. Of the 21 semantics-preserving operators, we have used approximately two-thirds. The operators we have not used are abrige, detour, unchain, chain, eliminate, import, rassoc, and lassoc. The first two operators delete or introduce a reflexive production which we have not encountered or needed in the Modelica grammar versions. The operator unchain is a version of inlining and applicable if the production is only defined by one nonterminal and the defining nonterminal occurs only in this production. We could have used it, for example, when inlining the additive expressions in the arithmetic expressions in JModelica with the same result. The operator chain reverses such an unchaining operation. As we have never come to a point where a defined nonterminal has not been in use anymore, we have not employed the operator eliminate. The operator import imports a grammar which did not occur in our scenario. The transformation operators rassoc and lassoc replace an iterative production by a right or left associative repeating equivalent of the former. As we have transformed OpenModelica and JModelica towards the Modelica specification, which does not include such associative productions, the operators have not been used.
The patterns in a production allow a handful of transformations. Similar patterns call for similar transformations. In most cases, the chain of transformations needed to achieve a conversion of a production towards the targeted production is straightforward. Renamings dissolve nominal mismatches and varying styles for terminal symbols. In the case of OpenModelica, simple renamings of terminals has lead to eight converging productions. Recursive productions need to change into iterative productions which calls for the use of deyaccification. Similar structures can be identified between colliding definitions, the deviations then guide the transformations. Different positions of optionals or parts of sequences are dissolved by corresponding inlining and extracting. Some inlining or unchaining is necessary if constructs are modeled using more productions than the objective grammar, as has been the case with access expressions in JModelica.

5.2.3 Order of the Application and Correctness of the Result

The order of the transformations within one XBGF file is determined by the intended effect on the production. The order between the different XBGF files is theoretically arbitrary. If the files impact different productions, the order does not matter. If the files impact the same productions, the order may not be arbitrary depending on the transformations. The reason for this dependence lays in the way of programming certain transformations. Unfortunately, only a handful transformations are fully automated. In case, a transformation needs the expression that should be changed and the expression to what the former should be transformed, conflicts are possible. For example, the operator \texttt{factor} needs the changing expression and the expression to which the changing expression should be transformed specified. If the production affected has been changed by a renaming in the beginning and the new name is specified in the XBGF file for the factorizing transformation, we cannot change the order of the transformations as we would need to change the name in the XBGF file to the previous one. The effect on the production would be the same, whether we would apply a factorization first or a renaming. Thus, the order of application of the transformations does not matter in theory. The XBGF files might need to change as we might refer to expressions in the grammar that do not exist as such yet. The transformer would take the first expression and search for occurrences, which it would not be able to find, and therefore, replace nothing. This small example of a scenario where the order cannot simply be interchanged is a representative for any more complex scenarios of transformations where it might not be possible to change the order of application.

In most cases in our comparisons, the transformations programmed lead to the targeted version in a direct way. No shorter chain of transformations exist. Let us take a look at the names and component references of JModelica to investigate the transformations used. Programming the transformations for the comparison of the Modelica specification and JModelica, we had unfolded the definition of names into the component references. The component references had been defined solely by names. We could not have used the transformation operator \texttt{unchain} as the names did not occur exclusively in the definition of the component references. We could not have inlined the names for the a similar reason. The names did occur in other places of the grammar though. We would have replaced every instance of the nonterminal \texttt{name} with its definition, not just the one occurrence in the component references. We contemplated equating names and component references. For the time being, let us assume we want to keep only one of the two constructs. Which one does not matter, as we can rename one into the other at the end. As we have unfolded the names into the component references, we would have needed to equate the productions which would have lead to the replacement of one nonterminal with the other and the deletion of the former production. In a second alternative, we could also have used the transformation \texttt{inline}, as described before, and then folded the component references in all the places where its definition occurred. A third alternative is to unfold the definition of component
references, i.e. name, in the whole grammar and eliminate the production component_reference afterwards. For this way, we would actually have needed the operator eliminate, an operator unused in our work. In summary, in this scenario of merging names and component references, the transformation chain of unfold(name in component_reference); equate(); leads to the same result as the chain of inline(name); fold(component_reference); and the chain of unfold(component_reference); eliminate(component_reference);. But this scenario is a rare one as in most cases, we deal with the remodeling of structures and not the elimination of a production. We can conclude that the transformations applied lead to a common result and our result is indeed correct.

The last section of this chapter concentrates on observations made using LCI for comparing Modelica grammar versions.

5.3 Experience with LCI

LCI provides an infrastructure to the task of comparing two grammar versions. It allows for simple programmable transformations. The transformations are partly automated and applied to the grammars in an automated fashion. The infrastructure provides the user with a guideline to programming transformations in the attempt to converge different grammar sources. Apart from the core task, the SLPS provides the grammar engineer with several tools to visualize the work, e.g., scripts to convert a grammar into a parse tree or to pretty-print BGF grammars and XBGF transformations into a more readable format. The benefits of using a published approach is that the method has already been tested and approved. As the publishers state in various papers and as can be seen in their ongoing work on the topic [10, 36, 12, 35], the work is not finished and starting points for improvements of the tools exist. Working with the tool reveals some of those starting points. We continue with a brief look at a handful of those starting points. The points concern the tool in general, the setup and the output of the tool as well as enhancements of the transformations. Many of these ideas translate into future work.

A very general objection is the necessity of the internal representation format. Though the argumentation why the use of the BGF is beneficial is conclusive, the use requires a user to become proficient to a certain stage in grammar extraction and conversion into BGF. As the BGF allows comparing different software artifacts with features that exceed the representation capability of textbook-style EBNF of context-free syntax, the additional effort pays off.

From the general objection regarding BGF, we move to more specific ideas of improvement. From the standpoint of someone external to the development of the tool, the work lacks a comprehensive documentation or a user’s guide. Though the introduction to the SLPS claims that every topic within the suite is commented to the point that it should be possible to run the programs with the help of existing Makefiles, the setup of the tool and the start of the work is not fluent. To identify the necessary libraries and tools for LCI to be able to run takes time. To get an idea how to use the LCI, one needs to consult several papers. For the LCI to work, some folders need to exist, something that is not clearly visible from the error reports shown. A small introduction to the setup of the tool would be helpful. A comprehensive language manual exists for XBGF but a user’s guide on how to start and what to keep in mind would facilitate the initial steps within the LCI.

The next idea deals with an inconvenience one might have noticed during Chapter 4. Whenever we want more detailed information about the differences between two grammars, we need to call the comparator script directly. The LCI outputs status lines about the overall success of the different steps within the LCI. For the result of the comparison, it creates a file with a list of the nonterminals that are shared but collide in their respective definitions. In addition,
the generated diagrams at the end of the LCI might give clues about the differences. But the output of the comparator is not completely saved. Instead, it is discarded within the LCI. These comparator outputs contrast the colliding definitions and list the exclusive nonterminals. Even the definitions of the exclusive nonterminals could be of interest. Currently, they are not part of the output. This information can help in programming the transformations required for aligning the grammars without the need to always look up the definitions in different documents.

After these suggestions concerning the LCI, the transformations receive attention. We revisit the grammar transformations and discuss scenarios that present a starting point for further automation of transformations and for suggesting transformations to the user of the LCI. Comparing the Modelica grammar versions needs mainly the transformations rename, factor and massage, vertical and deyaccify, extract, and inline. A simple example of ideas to automate transformations is the following. Similar situations call for identical sequences of transformations. A common combination has been the transformations vertical and deyaccify. Both need the nonterminal to whose production they are applied. The transformations could be combined to further automate the process. Another common scenario is the massaging of a choice of an epsilon and another expression into an optional expression. A script could take the nonterminal as input and convert all choices involving an epsilon into an optional. If this choice consists of more than two alternatives, the automated version might convert the alternatives into an optional choice expression consisting of all alternatives. A manually programmed massaging transformation would be needed if only parts of the alternatives should be transformed into an optional.

The transformation massage includes many possible transformations, which all can become tiresome to write as the transformation requires the expression to be transformed and the transformed version. In a complex grammar, these expressions may be rather large. The transformation factor faces a similar situation. A further automation of the programming of these transformations would be favorable. With the transformations massage and factor, we enter new scenarios, which lead to further ideas of improvement. These ideas would require even more work than the small automation approaches described previously. Given a certain pattern in the production, a handful of productions are applicable and reasonable. How to classify transformations as reasonable by a script may be even harder, but as a first step it might be feasible to suggest transformations for a production. As a starting point, the scope of the transformations suggested can be restricted to semantics-preserving transformations, and a tool or script outside of the LCI can take the grammar and the production name as input and suggest transformations for that production as output. For example, in case of a sequence with an inner choice, the tool can suggest a distribution. If the alternatives of a choice end on an identical expression, a factorization is applicable. The aforementioned transformation massage offers many possibilities for transformation. If programming a yaccification or deyaccification, the transformer already checks whether the input matches a “yaccify” pattern and replaces the input with the matching expression according to the pattern. In this case, code already exists to detect patterns. The basic idea behind suggestions of transformations is to detect patterns and suggest transformations applicable to these patterns. In a later stage, the tool might also generate the transformations, which would contribute to the efforts of further automation of the convergence process.

The considerations of further automation of language convergence marks the endpoint of this chapter on the evaluation of the results of the language convergence process. We have seen the implications of the differences on the languages generated by the grammars and the use of various simulation environments. Working with the LCI has lead to concrete ideas for improvement and future work. The body of the thesis is completed and we finish with a conclusion and a summary on future work.
Chapter 6

Conclusion

6.1 Future Work

The language convergence process of the Modelica expressions in the grammars of the Modelica specification, the OpenModelica parser description, and the JModelica parser description results in several new questions and possible tasks. These questions and tasks regard both Modelica and the LCI. We start with the future work concerning Modelica and close with propositions for future work for the LCI.

First, we would be interested in an LCI of the entire Modelica grammar, not just an excerpt of it. We could expand the statement concerning the relation of the grammars to the whole grammar and not only the expressions. A new LCI with a complete grammar can reveal new differences where programmers need to be careful if switching simulation environments. It might also help to further classify the difference between the OpenModelica grammar and the other two grammars about the referenced code and match expressions within the production `expression`. Additionally, we can extend the grammar sources by other commonly used simulation environments.

The encountered differences as a basis, maintainers and developers of the simulation environment could examine whether the differences are intended, and therefore, should be kept, or whether these differences are unintended and should be eliminated. If the differences are maintained, the user’s guide or manual could be updated to point to the areas where the simulation environment deviates from the Modelica specification.

We move from the field of grammar convergence to another area. Analyzing a grammar for a programming language almost automatically poses the question whether the grammar represents the language in a coherent way. We have already seen in the analysis of the differences that some productions represent constructs in a more precise way than others. One example are the names in Modelica in comparison to the component references in the other grammar versions. Another example is the way of modeling the function `initial()`. In a similar question, one might ask how much of what the grammar or the parser allows is accepted by a compiler. We have observed that the Modelica grammar allows a list of commas in the generation of output expression lists. A compiler might reject such an expression. Differences might always be present between what a parser and a compiler accepts. Nonetheless, a coherence should exist between a parser and a compiler just like between a parser and a language specification. A starting point for such an analysis would be the Modelica grammar, the point of reference for the simulation environments. The next paragraph focuses on ideas of improvement for the LCI.

We have discussed topics for future work in the context of the LCI in the previous chap-
6.2 Conclusion

This thesis aims to answer the question whether different Modelica grammar versions agree. The grammar sources subject to investigation are the Modelica language specification, the OpenModelica parser, and the JModelica parser. The time frame of a Bachelor thesis allows only for the section “Expressions” of the Modelica specification and the corresponding productions in JModelica and OpenModelica to be part of the investigation. The means to answer the question is to compare the grammars. Based on the encountered differences, we transform the grammars. The process forms a compare/transform cycle until we cannot dissolve any more differences. The tool we have chosen to support us in our investigation is the Language Convergence Infrastructure developed by Ralf Lämmel and Vadim Zaytsev. To accommodate the workings of the tool, we have written extractors that convert the grammar sources in the internal representation format BGF. Fed with the corresponding configuration, the tool outputs a verdict on the convergence of the grammar sources. A framework of programmable grammar transformations is associated with the tool LCI. The framework contains transformations applicable to the BGF grammars. For the grammar sources to converge, we have programmed XBGF transformations. With the help of the LCI and the XBGF, we have aligned large parts of the grammars and carved out the remaining differences.

The result of the LCIs is that all grammar versions disagree and that no version is strictly more permissive or restrictive than the other versions. In summary and neglecting some differences, the grammars form the relation of OpenModelica being more permissive than Modelica being more permissive than JModelica. The differences result into several situations where a programmer needs to revise their code when presenting code adhering to the Modelica grammar to the simulation environments. In addition, one needs to exercise caution if switching from one simulation environment to the other.

Future work includes the extension of the comparison to the entire grammar and to other simulation environments, an analysis of the representativeness of the grammar for the language, and possible actions concerning the encountered differences. The LCI can be enhanced by a user’s guide, a more detailed output, and further automation activities.
Appendix A

Grammar Sources

A.1 Modelica Specification

expression : simple_expression
            | "if" expression "then" expression { "elseif" expression "then" expression }
            | "else" expression

simple_expression : logical_expression [ ":" logical_expression [ ":" logical_expression ] ]

logical_expression : logical_term { "or" logical_term }

logical_term : logical_factor { "and" logical_factor }

logical_factor : [ "not" ] relation

relation : arithmetic_expression [ rel_op arithmetic_expression ]

rel_op : "<" | "<=" | "<" | ">=" | "==" | "<>"

arithmetic_expression : [ add_op ] term { add_op term }

add_op : "+" | "-" | ".+" | ".-"

term : factor { mul_op factor }

mul_op : "/" | ".*" | "/.*"

factor : primary [ ("." | "." ) primary ]

primary : "UNSIGNED_NUMBER"
         | "STRING"
         | "false"
         | "true"

( name | "der" | "initial" ) function_call_args

component_reference

[ "output_expression_list "]

[ "expression_list { ":" expression_list } "]

[ "function_arguments "]

end

name : [ "." ] "IDENT" { "." "IDENT" }

component_reference :
Listing A.1: Modelica Specification Expressions

A.2 JModelica

array_subscripts: "LBRACK" subscript_list "RBRACK"
subscript_list: subscript
| subscript_list "COMMA" subscript
subscript: "COLON" | exp
annotation: "ANNOTATION" class_modification
exp: simple_expression
| if_exp
| if_exp: "IF" exp "THEN" exp else_if_exp
else_if_exp: "ELSEIF" exp "THEN" exp else_if_exp
| "ELSE" exp
simple_expression:
| log_exp
| log_exp "COLON" log_exp
| log_exp "COLON" log_exp "COLON" log_exp
log_exp: log_exp "OR" log_exp
| log_exp "AND" log_exp
| "NOT" relation
relation:
| artm_exp "LT" artm_exp
A.2. JMODELICA

```
| artm_exp | "LEQ" artm_exp |
| artm_exp | "GT" artm_exp |
| artm_exp | "GEQ" artm_exp |
| artm_exp | "EQ" artm_exp |
| artm_exp | "NEQ" artm_exp |
| artm_exp | additive_exp |

additive_exp:
  | multiplicative_exp |
  | additive_exp | "PLUS" multiplicative_exp |
  | additive_exp | "MINUS" multiplicative_exp |
  | additive_exp | "DOTPLUS" multiplicative_exp |
  | additive_exp | "DOTMINUS" multiplicative_exp |

multiplicative_exp:
  | unary_exp |
  | multiplicative_exp | "MULT" unary_exp |
  | multiplicative_exp | "DIV" unary_exp |
  | multiplicative_exp | "DOTMULT" unary_exp |
  | multiplicative_exp | "DOTDIV" unary_exp |

unary_exp:
  | pow_exp |
  | "MINUS" unary_exp |
  | "PLUS" unary_exp |

pow_exp:
  | primary |
  | primary | "POW" primary |
  | primary | "DOTPOW" primary |

primary:
  | "UNSIGNED" |
  | "UNSIGNED_INTEGER" |
  | "TRUE" |
  | "FALSE" |
  | "STRING" |
  | access_expression |
  | der_expression |
  | LPAREN exp RPAREN |
  | function_call |
  | LBRAKC matrix RBRAKC |
  | LBRAKC function_arguments RBRAKC |
  | TIME |
  | END |

function_call:
  | parse_access LPAREN function_arguments RPAREN |

partial_function_call:
  | "FUNCTION" parse_access LPAREN function_arguments RPAREN |

function_arguments:
  | exp FOR for_indices |
  | arg_List_p |
  | arg_List_p COMMA named_arguments |
  | named_arguments |

named_arguments:
  | named_argument |
  | named_arguments COMMA named_argument |

function_argument_exp:
  | exp |
  | partial_function_call |

named_argument:
  | named_argument |
  | named_arguments EQUALS named_argument |

matrix:
```
### Listing A.2: JModelica Expressions

```
matrix_row
| matrix "SEMICOLON" matrix_row
matrix_row:
  exp
arg_list_p:
  function_argument_exp
| arg_list_p "COMMA" function_argument_exp
comment:
  string_comment? annotation?
string_comment:
  "STRING"
| string_comment "PLUS" "STRING"
access_expression:
parse_access
  der_expression:
    "DER" "LPAREN" exp "RPAREN"
parse_access:
  parse_access_loc
| "DOT" first_class_access
parse_access_loc:
  parse_access_single
| parse_access_single "DOT" parse_access_loc
first_class_access:
  class_access_single
| class_access_single "DOT" parse_access_loc
parse_access_single:
  "ID" array_subscripts?
class_access_single:
  "ID"

array_subscripts:
  "[" subscript ("," subscript)* "]
subscript:
  ":" | expression
annotation:
  "annotation" class_modification
expression:
  simple_expression
| "if" expression "then" expression ("elseif" expression "then" expression)* "else" expression
simple_expression:
  logical_expression (";" logical_expression (";" logical_expression)?)?
logical_expression:
  logical_factor ("or" | "and") logical_expression*
relation:
  arithmetic_expression (rel_op arithmetic_expression)?
arithmetic_expression:
  term (add_op term)*
term:
  factor (mul_op factor)*
factor:
  ("-" | "+")* primary ("-" | "+") primary?
primary:
  "UNSIGNED_NUMBER"
| "true"
| "false"
| "STRING"
| component_reference
| "der" (" expression ")
```

---

APPENDIX A. GRAMMAR SOURCES

---
Listing A.3: Transformed JModelica Expressions

```
| "(" output_expression_list ")"
| name function_call_args
| "(" expression_list (":" expression_list) "")"
| ["(" function_arguments ")"
| "end"

output_expression_list:
  expression
function_call_args:
  
function_arguments:
  expression "for" for_indices
  function_argument ("," function_argument)"
  function_argument ("," function_argument) "," named_arguments
  named_arguments
  named_argument ("," named_arguments)"

function_argument:
  expression
  "function" name (" function_arguments? ")"

named_argument:
  "IDENT" == function_argument
expression_list:
  expression ("," expression)*

comment:
  string_comment annotation?

component_reference:
  "IDENT" array_subscripts? ("." "IDENT" array_subscripts?)"
  
name:
  "IDENT" array_subscripts? ("." "IDENT" array_subscripts?)"
  
rel_op:
  "<" | "<=" | ">" | ">=" | "==" | "<>"
add_op:
  "-" | "+" | ".+" | ",-"
mul_op:
  "*" | "/" | ".*" | "/."

string_comment:
  ("STRING" ("+" "STRING")*)

logical_factor:
  "not"? relation

arithmetic_expression (rel_op arithemetic_expression)?
arithmetic_expression:
```

A.2. JMODELICA
APPENDIX A. GRAMMAR SOURCES

Listing A.4: Transformed JModelica Expressions after LCI with OpenModelica
A.3 OpenModelica

expression:
  if_expression
  | simple_expression
  | code_expression
  | part_eval_function_expression
  | match_expression

part_eval_function_expression:
  "FUNCTION" component_reference function_call

if_expression:
  "IF" expression "THEN" expression elseif_expression_list? "ELSE" expression
elseif_expression_list:
  elseif_expression elseif_expression_list?
elseif_expression:
  "ELSEIF" expression "THEN" expression

simple_expression:
  simple_expr ("COLONCOLON" simple_expression)?
  | "IDENT" "AS" simple_expression

simple_expr:
  logical_expression ("COLON logical_expression ("COLON logical_expression)")?

logical_expression:
  logical_term ("T\(\text{OR}\) logical_term")

logical_term:
  logical_factor ("T\(\text{AND}\) logical_factor")

logical_factor:
  "T\(\text{NOT}\)? relation

relation:
  arithmetic_expression (("LESS" | "LESSEQ" | "GREATER" | "GREATEREQ" | "EQEQ" | "LESSGT") arithmetic_expression)?

arithmetic_expression:
  unary_arithmetic_expression ("PLUS" | "MINUS" | "PLUS\text{EW}" | "MINUS\text{EW}") term*

unary_arithmetic_expression:
  "PLUS" term
  | "MINUS" term
  | "PLUS\text{EW}" term
  | "MINUS\text{EW}" term

term:
  factor (("STAR" | "SLASH" | "STAR\text{EW}" | "SLASH\text{EW}") factor)*

factor:
  primary (("POWER" | "POWER\text{EW}") primary)?

primary:
  "UNSIGNED\text{INTEGER}"
  | "UNSIGNED\text{REAL}"
  | "STRING"
  | "T\(\text{FALSE}\"
  | "T\(\text{TRUE}\"
  | component_reference_function_call
  | "DER" function_call
  | "LPAR" output_expression_list
  | "LBRACK" matrix_expression_list "RBRACK"
  | "LBRACE" for_or_expression_list "RBRACE"
  | "T\(\text{END}\"

matrix_expression_list:
  expression_list ("SEMICOLON" matrix_expression_list)?

component_reference_function_call:
  component_reference_function_call
  | "INITIAL" "LPAR" "RPAR"

component_reference:
  "DOT"? component_reference2
Listing A.5: OpenModelica Expressions

expression:
  "if" expression "then" expression ("elseif" expression "then" expression)* "else" expression
  | (simple_expression ":=") | ("IDENT" "as")* simple_expression
  | code_expression
  | part_eval_function_expression
  | match_expression

part_eval_function_expression:
  "function" name function_call_args

simple_expression:
  logical_expression (":=" logical_expression (":=" logical_expression)?)
logical_expression:
  logical_term ("or" logical_term)*
logical_term:
  logical_factor ("and" logical_factor)*

logical_factor:
  "not"? relation
relation:
  arithmetic_expression (rel_op arithmetic_expression)?

arithmetic_expression:
  add_op? term (add_op term)*
term:
  factor (mul_op factor)*
A.3. OPENMODELICA

Listing A.6: Transformed OpenModelica Expressions

expression:
  "if" expression "then" expression ("elseif" expression "then" expression)* "else" expression
  | ((simple_expression "::") | ("IDENT" "as"))* simple_expression
APPENDIX A. GRAMMAR SOURCES

| code_expression
| part_eval_function_expression
| match_expression
part_eval_function_expression:
  "function" component_reference function_call_args
simple_expression:
  logical_expression (":" logical_expression (":" logical_expression)?)?
logical_expression:
  logical_term ("or" logical_term)*
logical_term:
  logical_factor ("and" logical_factor)*
logical_factor:
  "not"? relation
relation:
  arithmetic_expression (rel_op arithmetic_expression)?
arithmetic_expression:
  add_op? term (add_op term)*
term:
  factor (mul_op factor)*
factor:
  primary ("" | ".") primary)?
primary:
  (component_reference | "der") function_call_args
  "UNSIGNED_NUMBER"
  "STRING"
  "false"
  "true"
  component_reference
  "initial" "(" ")"
  "(" output_expression_list ")"
  "(" expression_list ";" expression_list ")"
  "(" for_or_expression_list ")"
  "end"
component_reference:
  "."? ("IDENT" | "operator") array_subscripts? ("." ("IDENT" | "operator")
array_subscripts?)*
function_call_args:
  "(" function_arguments? ")"
for_or_expression_list:
  (function_argument ("","" function_argument ",")* function_argument?)
  ("for" for_indices)?
named_arguments:
  named_argument ("," named_arguments)?
named_argument:
  ("IDENT" | "operator") "+" function_argument
expression_list:
  expression ("","" expression)*
array_subscripts:
  "(" subscript ("," subscript)* ")"
subscript:
  expression | ".:"
comment:
  string_comment annotation?
string_comment:
  ("STRING" ("+" "STRING")*)?
annotation:
  "annotation" class_modification
rel_op:
  "<" | "<=" | ">" | ">=" | ">=" | ">>
mul_op:
  "*" | "/" | ".*" | "/."
add_op:
output_expression_list:
  expression? ("," expression?)
  function_arguments:
    named_arguments
    | function_argument (""," (function_argument ",")* function_argument?) | ("for" for_indices))? named_arguments?
function_argument:
  expression
## Appendix B

### BGF

#### B.1 BGF Grammar

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammar:</td>
<td>root :: nonterminal* production*</td>
</tr>
<tr>
<td>production:</td>
<td></td>
</tr>
</tbody>
</table>
  - label :: label? nonterminal :: nonterminal expression |
| expression: | 
  - epsilon :: ε |
| | 
  - empty :: ε |
| | 
  - value :: value |
| | 
  - any :: ε |
| | 
  - terminal :: terminal |
| | 
  - nonterminal :: nonterminal |
| | 
  - selectable :: (selector :: selector expression) |
| | 
  - sequence :: (expression+) |
| | 
  - marked :: expression |
| | 
  - choice :: (expression+) |
| | 
  - optional :: expression |
| | 
  - plus :: expression |
| | 
  - star :: expression |
| value: | 
  - int :: ε |
| | 
  - string :: ε |
| label: | STR |
| nonterminal: | STR |
| selector: | STR |
| terminal: | STR |

Listing B.1: BGF Grammar
B.2 Example Grammar

```
<?xml version="1.0" encoding="UTF-8"?>
<bgf:grammar xmlns:bgf="http://planet-sl.org/bgf">
  <bgf:production>
    <nonterminal>subscript</nonterminal>
    <bgf:expression>
      <choice>
        <bgf:expression>
          <nonterminal>expression</nonterminal>
        </bgf:expression>
        <terminal>::</terminal>
      </choice>
      <bgf:expression>
        <nonterminal>expression</nonterminal>
      </bgf:expression>
    </bgf:expression>
  </bgf:production>
</bgf:grammar>
```

Listing B.2: Example Modelica Specification BGF

```
<?xml version="1.0" encoding="UTF-8"?>
<bgf:grammar xmlns:bgf="http://planet-sl.org/bgf">
  <bgf:production>
    <nonterminal>subscript</nonterminal>
    <bgf:expression>
      <choice>
        <bgf:expression>
          <terminal>::</terminal>
        </bgf:expression>
        <nonterminal>exp</nonterminal>
      </choice>
    </bgf:expression>
  </bgf:production>
</bgf:grammar>
```

Listing B.3: Example JModelica BGF
## Appendix C

### LCF

#### C.1 LCF Grammar

<table>
<thead>
<tr>
<th>Scenario:</th>
<th>shortcut* tools source+ target+ testset*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortcut:</td>
<td>name::id expansion::xstring</td>
</tr>
<tr>
<td>Tools:</td>
<td>transformer::tool comparator::tool validator::tool? generator*</td>
</tr>
<tr>
<td>Tool:</td>
<td>grammar::xstring tree::xstring?</td>
</tr>
<tr>
<td>Generator:</td>
<td>name::id command::xstring</td>
</tr>
<tr>
<td>Testset:</td>
<td>name::id command::xstring</td>
</tr>
<tr>
<td>Source:</td>
<td>name::id derived? source-grammar source-tree? test-set::refid*</td>
</tr>
<tr>
<td>Derived:</td>
<td>from::refid using::string</td>
</tr>
<tr>
<td>Source-grammar:</td>
<td>extraction::xstring parsing::xstring? evaluation::xstring?</td>
</tr>
<tr>
<td>Source-tree:</td>
<td>extraction::xstring evaluation::xstring?</td>
</tr>
<tr>
<td>Target:</td>
<td>name::id branch+</td>
</tr>
<tr>
<td>Branch:</td>
<td>input::refid preparation::phase? nominal-matching::phase? structural-matching::phase? (extension::phase</td>
</tr>
<tr>
<td>Phase:</td>
<td>step::{perform-transformation::string</td>
</tr>
<tr>
<td>Automated-transformation:</td>
<td>method::refid result::string</td>
</tr>
</tbody>
</table>

Listing C.1: LCF Grammar
C.2 Example Grammar

```xml
<?xml version="1.0" encoding="UTF-8"?>
<lf:configuration xmlns:lf="http://planet-sl.org/lcf">
  <shortcut>
    <name>impl</name>
    <expansion>../</expansion>
  </shortcut>
  <shortcut>
    <name>srcs</name>
    <expansion>impl</expansion>/srcs</expansion>
  </shortcut>
  <shortcut>
    <name>extraction</name>
    <expansion>impl</expansion>/extraction</expansion>
  </shortcut>
  <shortcut>
    <name>slps</name>
    <expansion>...</expansion>/slps-master</expansion>
  </shortcut>
  <shortcut>
    <name>tools</name>
    <expansion>slps</expansion>/shared/tools</expansion>
  </shortcut>
  <tools>
    <transformer>
      <grammar>tools</grammar>/xhgf</transformer>
    </transformer>
    <comparator>
      <grammar>tools</grammar>/gdt</comparator>
  </tools>
  <source>
    <name>leaf-expr-j</name>
    <grammar>
      <extraction>
        <expand>extraction</expand>/j2bgf <expand>srcs</expand>/jmod/leaf-expr-j.txt
      </extraction>
    </grammar>
  </source>
  <source>
    <name>leaf-expr-m</name>
    <grammar>
      <extraction>
        <expand>extraction</expand>/m2bgf <expand>srcs</expand>/mod/leaf-expr-m.txt
      </extraction>
    </grammar>
  </source>
  <target>
    <name>node</name>
    <branch>
      <input>leaf-expr-j</input>
    </branch>
    <branch>
      <input>leaf-expr-m</input>
    </branch>
  </target>
</lf:configuration>
```
<lcf:configuration xmlns:lcf="http://planet-sl.org/lcf">
  <shortcut>
    <name>impl</name>
    <expansion>impl</expansion>
  </shortcut>
  <shortcut>
    <name>srcs</name>
    <expansion><expand>impl</expand>/srcs</expansion>
  </shortcut>
  <shortcut>
    <name>extracts</name>
    <expansion>extracts</expansion>
  </shortcut>
  <shortcut>
    <name>slps</name>
    <expansion>slps</expansion>
  </shortcut>
  <shortcut>
    <name>tools</name>
    <expansion>slps</expansion>/shared/tools
  </shortcut>
  <transformation>
    <grammar>tools</grammar>
    <transformer>xbgf</transformer>
  </transformation>
  <transformation>
    <grammar>tools</grammar>
    <transformer>gdt</transformer>
  </transformation>
  <source>
    <name>leaf-expr-j</name>
    <grammar>
      <extraction>
        <expand>extracts</expand>/j2bfg <expand>srcs</expand>/jmod/leaf-expr-j.txt
      </extraction>
    </grammar>
  </source>
  <source>
    <name>leaf-expr-m</name>
    <grammar>
      <extraction>
        <expand>extracts</expand>/m2bfg <expand>srcs</expand>/mod/leaf-expr-m.txt
      </extraction>
    </grammar>
  </source>
  <target>
    <name>node</name>
    <branch>
      <input>leaf-expr-j</input>
      <structural-matching>
        <perform>rename-j-leaf</perform>
      </structural-matching>
    </branch>
  </target>
</lcf:configuration>
APPENDIX C. LCF

Listing C.3: Updated Example LCF

C.3 Modelica Specification vs. JModelica

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<lf:configuration xmlns:lf="http://planet-sl.org/lcf">
  <shortcut>
    <name>impl</name>
  </shortcut>
  <shortcut>
    <name>srs</name>
  </shortcut>
  <shortcut>
    <name>slps</name>
  </shortcut>
  <shortcut>
    <name>tools</name>
  </shortcut>
  <tools>
    <transformer>
      <grammar>
        <expand>tools/xbgf/grammar</expand>
      </grammar>
    </transformer>
    <transformer>
      <grammar>
        <expand>tools/gdt/grammar</expand>
      </grammar>
    </transformer>
  </tools>
  <source>
    <name>expr-j</name>
    <extract>
      <expand>expression</expand>/j2bgf <expand>srs</expand>/jmod/expr-j.txt
    </extract>
  </source>
  <source>
    <name>expr-mc</name>
    <extract>
      <expand>expression</expand>/m2bgf <expand>srs</expand>/mod/expr-m.txt
    </extract>
  </source>
</lf:configuration>
```
Listing C.4: Initial LCF for the Modelica Specification vs. JModelica

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<lcf:configuration xmlns:lcf="http://planet-sl.org/lcf">
  <shortcut>
    <name>impl</name>
    <expansion>../</expansion>
  </shortcut>
  <shortcut>
    <name>srcs</name>
    <expansion>impl</expansion>/srcs</expansion>
  </shortcut>
  <shortcut>
    <name>extrac</name>
    <expansion>impl</expansion>/extrac</expansion>
  </shortcut>
  <shortcut>
    <name>slps</name>
    <expansion>../slps-master</expansion>
  </shortcut>
  <shortcut>
    <name>tools</name>
    <expansion>/slps</expansion>/shared/tools</expansion>
  </shortcut>
  <shortcut>
    <name>tools</name>
    <grammar><expand>tools</expand>/xbgf</grammar>
  </shortcut>
  <shortcut>
    <grammar><expand>tools</expand>/gdt</grammar>
  </shortcut>
  <source>
    <name>expr-j</name>
    <grammar><expand>extrac</expand>/j2bgf <expand>srcs</expand>/jmod/expr-j.txt</grammar>
  </source>
  <source>
    <name>expr-m</name>
    <grammar><expand>extrac</expand>/m2bgf <expand>srcs</expand>/mod/expr-m.txt</grammar>
  </source>
</lcf:configuration>
```
Listing C.5: Final LCF for the Modelica Specification vs. JModelica

C.4 Modelica Specification vs. OpenModelica
Listing C.6: Initial LCF for the Modelica Specification vs. OpenModelica

```xml
<?xml version="1.0" encoding="UTF-8"?>
<lcf:configuration xmlns:lcf="http://planet-sl.org/lcf">
  <shortcut>
    <name>impl</name>
    <expansion>...</expansion>
  </shortcut>
  <shortcut>
    <name>src</name>
    <expansion>impl</expansion>/srcs</expansion>
  </shortcut>
  <shortcut>
    <name>extraction</name>
    <expansion>impl</expansion>/extraction</expansion>
  </shortcut>
  <shortcut>
    <name>slps</name>
    <expansion>...</expansion>/slps-master</expansion>
</lcf:configuration>
```
<shortcut/>
  <shortcut>
    <name>tools</name>
    <expansion>expand</expansion>/shared/tools/<expansion>
  </shortcut>

<tools>
  <transformer>
    <grammar>expand</grammar>/xbgf/grammar
  </transformer>
  <transformer>
    <grammar>expand</grammar>/gdt/grammar
  </transformer>
</tools>

<source>
  <name>expr-oc</name>
  <grammar>
    <extraction>
      <expand>extraction</expand>/o2bgf <expand>srcs</expand>/omod/expr-oc.g
    </extraction>
  </grammar>
</source>

<source>
  <name>expr-mc</name>
  <grammar>
    <extraction>
      <expand>extraction</expand>/m2bgf <expand>srcs</expand>/mod/expr-m.txt
    </extraction>
  </grammar>
</source>

<target>
  <name>target</name>
  <branch>
    <input>expr-oc</input>
    <nominal-matching>
      <perform>renameT-oc</perform>
    </nominal-matching>
    <structural-matching>
      <perform>sub-list-inline</perform>
      <perform>exp-list-massage</perform>
      <perform>rel-op-extract</perform>
      <perform>mul-op-extract</perform>
      <perform>una-inline</perform>
      <perform>ifexp-inline</perform>
      <perform>matrix-inline</perform>
      <perform>ref-align</perform>
      <perform>for-or-align</perform>
      <perform>fct-align</perform>
      <perform>call-inline</perform>
      <perform>part</perform>
      <perform>smp-inline</perform>
      <perform>eliminate-oc</perform>
    </structural-matching>
  </branch>
  <branch>
    <input>expr-mc</input>
  </branch>
</target>

</lcf:configuration>
C.5 OpenModelica vs. JModelica

Listing C.7: Final LCF for the Modelica Specification vs. OpenModelica

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<lcf:configuration xmlns:lcf="http://planet-sl.org/lcf">
  <shortcut>
    <name>impl</name>
    <expansion>../</expansion>
  </shortcut>
  <shortcut>
    <name>bgf</name>
    <expansion><expand>impl</expand>/lcf/bgfc/expansion>
  </shortcut>
  <shortcut>
    <name>extraction</name>
    <expansion><expand>impl</expand>/extraction/expansion>
  </shortcut>
  <shortcut>
    <name>slps</name>
    <expansion>.././slps-master/shared/tools/expansion>
  </shortcut>
  <tools>
    <grammar><expand>tools</expand>/xbgf/grammar>
    <transformer>
      <comparator>
        <grammar><expand>tools</expand>/gdt/grammar>
        <source>
          <name>tgt-j</name>
          <grammar>
            <extraction>
              <expand>extraction</expand>/copybgf <expand>bgf</expand>/target-j.bgf
            </extraction>
          </grammar>
          <source>
            <name>tgt-o</name>
            <grammar>
              <extraction>
                <expand>extraction</expand>/copybgf <expand>bgf</expand>/target-o.bgf
              </extraction>
            </grammar>
          </source>
        </comparator>
      </transformer>
    </grammar>
  </tools>
  <target>
    <name>target</name>
    <branch>
      <input>tgt-j</input>
    </branch>
    <branch>
      <input>tgt-o</input>
    </branch>
  </target>
</lcf:configuration>
```
Listing C.8: Initial LCF for OpenModelica vs. JModelica

```xml
<?xml version="1.0" encoding="UTF-8"?>
<lcf:configuration xmlns:lcf="http://planet-sl.org/lcf">
  <shortcut>
    <name>impl</name>
    <expansion>..</expansion>
  </shortcut>
  <shortcut>
    <name>bgf</name>
    <expansion>impl</expansion>/lcf/bgf</expansion>
  </shortcut>
  <shortcut>
    <name>extraction</name>
    <expansion>impl</expansion>/extraction</expansion>
  </shortcut>
  <shortcut>
    <name>slps</name>
    <expansion>...</expansion>/slps-master</expansion>
  </shortcut>
  <shortcut>
    <name>tools</name>
    <expansion>slps</expansion>/shared/tools</expansion>
  </shortcut>
  <tools>
    <transformer>
      <grammar>tools</grammar>/xbgf</grammar>
    </transformer>
    <comparator>
      <grammar>tools</grammar>/gdt</grammar>
    </comparator>
  </tools>
  <source>
    <name>tgt−j</name>
    <grammar>
      <extraction>
        <expand>extraction</expand>/copybgf <expand>bgf</expand>/target−j.bgf
        <extraction>
      </grammar>
    </source>
    <source>
      <name>tgt−o</name>
      <grammar>
        <extraction>
          <expand>extraction</expand>/copybgf <expand>bgf</expand>/target−o.bgf
          <extraction>
        </grammar>
      </source>
    <target>
      <name>target</name>
      <branch>
        <input>tgt−j</input>
        <structural−matching>
        <perform>elim−name−j</perform>
        <perform>tgt−refactor</perform>
      </branch>
    </target>
</lcf:configuration>
```
<table>
<thead>
<tr>
<th>Listing C.9: Final LCF for OpenModelica vs. JModelica</th>
</tr>
</thead>
<tbody>
<tr>
<td>60</td>
</tr>
<tr>
<td>&lt;structural-matching&gt;</td>
</tr>
<tr>
<td>&lt;/branch&gt;</td>
</tr>
<tr>
<td>&lt;branch&gt;</td>
</tr>
<tr>
<td>&lt;input&gt;tgt-0&lt;/input&gt;</td>
</tr>
<tr>
<td>&lt;structural-matching&gt;</td>
</tr>
<tr>
<td>&lt;perform&gt;elim-name-o&lt;/perform&gt;</td>
</tr>
<tr>
<td>&lt;/structural-matching&gt;</td>
</tr>
</tbody>
</table>
Appendix D

Comparator Output

D.1 Modelica Specification vs. JModelica

Names of defined nonterminals differ:

- Intersection | array_subscripts, subscript, annotation, simple_expression, relation, primary, function_arguments, named_arguments, named_argument, comment, string_comment |
- bgf/expr-j.bgf only: [subscript_list, exp, if_exp, else_if_exp, log_exp, artm_exp, additive_exp, multiplicative_exp, unary_exp, pow_exp, function_call, partial_function_call, function_argument_exp, named_argument_id, matrix, matrix_row, arg_list_p, access_expression, der_expression, parse_access, parse_access_loc, first_class_access, parse_access_single, class_access_single ]
- bgf/expr-m.bgf only: [expression, logical_expression, logical_term, logical_factor, rel_op, arithmetic_expression, add_op, term, mul_op, factor, name, component_reference, function_call_args, function_argument, output_expression_list, expression_list ]

Comparisons per (common) nonterminal:

- Fail (1/1): array_subscripts.
  - [] : (t(LBRACK), n(subscript_list), t(RBRACK))
  vs.
  - [] : (t(ARRAY), n(subscript_list))
- Fail (1/1): subscript.
  - [] : (t(:), n(exp))
  vs.
  - [] : (t(ANNOTATION), n(class_modification))
- Fail (1/1): simple_expression.
  - [] : (n(log_exp), (n(log_exp), t(COLON), n(log_exp)), (n(log_exp), t(COLON), n(log_exp)))
  vs.
  - [] : (n(logical_expression), ?(t(:), n(logical_expression)))
- Fail (1/1): relation.
APPENDIX D. COMPARATOR OUTPUT

Listing D.1: Initial Comparator Output of the Modelica Specification vs. JModelica

- \[\ldots\] \{(\{[n(\text{arithmetic\_expression})], (\{[n(\text{rel\_op})], n(\text{arithmetic\_expression}))\} )\} \]
  \hspace{1cm} vs.
- \[\ldots\] \{(\{[n(\text{arithmetic\_expression})], (\{[n(\text{rel\_op})], n(\text{arithmetic\_expression}))\} )\} \]
  \hspace{1cm} Fail (1/1): primary.
- \[\ldots\] \{(t(\text{UNSIGNED\_NUMBER}), t(\text{UNSIGNED\_INTEGER}), t(\text{TRUE}), t(\text{FALSE}), t(\text{STRING}), n(\text{access\_expression}), n(\text{der\_expression})) \}, \{(t(\text{LPAREN}), n(\text{exp}), t(\text{RPAREN}))\}, \{(n(\text{function\_call})), \{(t(\text{LBRACK}), n(\text{matrix})), t(\text{RBRACK}))\}, \{(t(\text{LBRACE}), n(\text{function\_arguments})), t(\text{RBRACE}))\}, t(\text{TIME}), t(\text{END})\}
  \hspace{1cm} vs.
- \[\ldots\] \{(t(\text{UNSIGNED\_NUMBER}), t(\text{STRING}), t(\text{false}), t(\text{true})), \{(n(\text{name}), t(\text{der}), t(\text{initial})), n(\text{component\_reference}), \{(t()\}, n(\text{output\_expression\_list}), t(\text{()}))\}, \{(t()\}, n(\text{expression\_list}), \{*, \{t()\}, n(\text{expression\_list}\}, t(\text{()})\}, \{(t()\}, n(\text{function\_arguments})), t(\text{()})\}, t(\text{end})\}
  \hspace{1cm} Fail (1/1): function\_arguments.
- \[\ldots\] \{(n(\text{exp}), t(\text{FOR}), n(\text{for\_indices}))\}, n(\text{arg\_list\_p}), \{(n(\text{arg\_list\_p}), t(\text{COMMA}), n(\text{named\_arguments}))\}
  \hspace{1cm} Fail (1/1): named\_arguments.
- \[\ldots\] \{n(\text{named\_argument}), \{n(\text{named\_arguments}), t(\text{COMMA}), n(\text{named\_arguments})\}\}
  \hspace{1cm} vs.
- \[\ldots\] \{n(\text{named\_argument}), \{\{t()\}, n(\text{named\_arguments})\}\}
  \hspace{1cm} Fail (1/1): named\_argument.
- \[\ldots\] \{n(\text{named\_argument}), t(\text{EQUALS}), n(\text{function\_argument\_exp})\}
  \hspace{1cm} vs.
- \[\ldots\] \{n(\text{string\_comment}), \{n(\text{annotation})\}\}
  \hspace{1cm} Fail (1/1): comment.
- \[\ldots\] \{n(\text{string\_comment}), \{n(\text{annotation})\}\}
  \hspace{1cm} vs.
- \[\ldots\] \{n(\text{string\_comment}), \{n(\text{string\_comment}), t(\text{PLUS}), t(\text{STRING})\}\}
  \hspace{1cm} Fail (1/1): string\_comment.
- \[\ldots\] \{(t(\text{STRING}), \{\{t()\}, n(\text{string\_comment})\}\}
  \hspace{1cm} vs.
- \[\ldots\] \{(t(\text{STRING}), \{\{t()\}, t(\text{STRING})\}\})
  \hspace{1cm} Roots agree.

- Names of defined nonterminals differ.
  - Intersection \{array\_subscripts, subscript, annotation, expression, simple\_expression, logical\_expression, relation, arithmetic\_expression, term, factor, primary, output\_expression\_list, function\_call\_args, function\_arguments, named\_arguments, function\_argument, named\_argument, expression\_list, comment, component\_reference, name, rel\_op, add\_op, mul\_op, string\_comment, logical\_factor\}
  - bgf/expr->\text{rename}.2, subscr.\text{ifexp}.\text{smpexp}.rel.\text{extract}.\text{message}.\text{artm}.\text{extract}.\text{matrix}.access.named.fct.2.eliminate.refactor.bgf
  - bgf/expr-m.bgf

- virtualbox:˜/Dokumente/impl/ICF$../.slps-master/shared/tools/gdt bgf/expr
  - j.rename.2, subscr.\text{ifexp}.\text{smpexp}.\text{rel}.\text{extract}.\text{message}.\text{artm}.\text{extract}.\text{matrix}.access.named.fct.2.eliminate.refactor.bgf
  - Normalizing bgf/expr->\text{rename}.2, subscr.\text{ifexp}.\text{smpexp}.rel.\text{extract}.\text{message}.\text{artm}.\text{extract}.\text{matrix}.access.named.fct.2.eliminate.refactor.bgf
  - Normalizing bgf/expr-m.bgf

- Diffing bgf/expr->\text{rename}.2, subscr.\text{ifexp}.\text{smpexp}.rel.\text{extract}.\text{message}.\text{artm}.\text{extract}.\text{matrix}.access.named.fct.2.eliminate.refactor.bgf and bgf/expr-m.bgf

- VirtualBox:˜/Dokumente/impl/ICF$../.slps-master/shared/tools/gdt bgf/expr
- j.rename.2, subscr.\text{ifexp}.\text{smpexp}.\text{rel}.\text{extract}.\text{message}.\text{artm}.\text{extract}.\text{matrix}.access.named.fct.2.eliminate.refactor.bgf
D.1. MODELICA SPECIFICATION VS. JMODELICA

Comparisons per (common) nonterminal:

- Ok: array_subscripts.
- Ok: subscript.
- Ok: annotation.
- Ok: expression.
- Ok: simple_expression.
- Fail (1/1): logical_expression.
  - [] , ( [ [ n ( logical_factor ) , * ( , ( [ [ t ( or ) , t ( and ) ] ) ) , n ( logical_expression ) ] ) ] )
  vs.
  - [] , ( [ [ n ( logical_term ) , * ( , ( [ [ t ( or ) , n ( logical_term ) ] ) ) ] ) ] )
  - Ok: relation.
- Fail (1/1): arithmetic_expression.
  - [] , ( [ [ n ( term ) , * ( , ( [ [ t ( or ) , n ( term ) ] ) ) ] ) ] )
  vs.
  - [] , ( [ [ ? ( n ( add_op ) ) , n ( term ) , * ( , ( [ [ t ( or ) , n ( term ) ] ) ) ] ) ] )
  - Ok: term.
- Fail (1/1): factor.
  - [] , ( [ [ * ( ; ( [ t ( - ) , t ( + ) ] ) ) , n ( primary ) , ? ( , ( [ [ t ( ' ) , t ( . ' ) ] ) ) , n ( primary ) ) ] ) ] )
  vs.
  - [] , ( [ [ n ( primary ) , ? ( , ( [ [ t ( ' ) , t ( . ' ) ] ) ) , n ( primary ) ) ] ) ] )
  - Fail (1/1): output_expression_list.
  - [] , ( [ [ , ( [ t ( UNSIGNED_NUMBER ) , t ( true ) , t ( false ) , t ( STRING ) , n ( component_reference ) , t ( der ) , t ( ( ) , n ( output_expression_list ) , t ( ) ) ) , , ( [ t ( ( ) , n ( function_call_args ) , t ( ) ) ] ) , , ( [ t ( ( ) , n ( function_arguments ) , t ( ) ) ] ) , , ( [ t ( ) , n ( function_arguments ) , t ( ) ) ] ) , t ( end ) ] ) ] )
  vs.
  - [] , ( [ [ , ( [ t ( UNSIGNED_NUMBER ) , t ( STRING ) , t ( true ) , t ( false ) , , ( [ ; ( [ n ( name ) , t ( der ) , t ( initial ) ] ) , n ( component_reference ) , t ( der ) , t ( ( ) , n ( output_expression_list ) , t ( ) ) ) , , ( [ t ( ) , n ( function_call_args ) , t ( ) ) ] ) , , ( [ t ( ) , n ( function_arguments ) , t ( ) ) ] ) , t ( end ) ] ) ] )
  - Fail (1/1): function_call_args.
  - Ok: function_call_args.
  - Fail (1/1): named_arguments.
  - [] , ( [ [ , ( [ t ( function ) , n ( name ) , t ( ) , ? ( n ( function_arguments ) ) ] ) ] ) ] )
  vs.
  - [] , ( [ [ , ( [ t ( function ) , n ( name ) , t ( ) , ? ( n ( named_arguments ) ) ] ) , n ( expression ) ] ) ] )
  - Ok: named_arguments.
  - Ok: expression_list.
  - Ok: comment.
- Fail (1/1): component_reference.
  - [] , ( [ [ , ( [ t ( IDENT ) , ? ( n ( array_subscripts ) ) , * ( , ( [ t ( ) , t ( IDENT ) , ? ( n ( array_subscripts ) ) ] ) ) ] ) , , ( [ t ( ) , t ( IDENT ) , * ( , ( [ t ( ) , t ( IDENT ) , ? ( n ( array_subscripts ) ) ] ) ) ] ) ) ] )
  vs.
  - [] , ( [ [ , ( [ t ( ) , t ( IDENT ) , ? ( n ( array_subscripts ) ) , * ( , ( [ t ( ) , t ( IDENT ) , ? ( n ( array_subscripts ) ) ] ) ) ] ) ] )
  - Fail (1/1): name.
  - [] , ( [ [ , ( [ t ( IDENT ) , ? ( n ( array_subscripts ) ) , * ( , ( [ t ( ) , t ( IDENT ) , ? ( n ( array_subscripts ) ) ] ) ) ] ) ] )
  - Fail (1/1): identifier.
APPENDIX D. COMPARATOR OUTPUT

Listing D.2: Final Comparator Output of the Modelica Specification vs. JModelica

D.2 Modelica Specification vs. OpenModelica

```
D.2 Modelica Specification vs. OpenModelica

vm@vm-VirtualBox:~/Dokumente/impl/lfcS ././slps-master/shared/tools/gdt bgf/expr
-o.bgf bgf/expr-m.bgf
Normalizing bgf/expr-o.bgf.
Normalizing bgf/expr-m.bgf.
Differ bgf/expr-o.bgf and bgf/expr-m.bgf.

- Names of defined nonterminals differ.
  - Intersection [expression, simple_expression, logical_expression, logical_term, logical_factor, relation, arithmetic_expression, term, factor, primary, component_reference, function_arguments, named_arguments, named_argument, output_expression_list, expression_list, array_subscripts, subscript, comment, string_comment, annotation].
  - bgf/expr-o.bgf only: [part_eval_function_expression, if_expression, else_expression_list, elseif_expression, simple_expr, unary_arithmetic_expression, matrix_expression_list, component_reference, function_call, component_reference2, function_call, for_or_expression_list, for_or_expression_list2, subscript_list].
  - bgf/expr-m.bgf only: [rel_op, add_op, mul_op, name, function_call, args, function_argument].

Comparisons per (common) nonterminal:

- Fail (1/1): expression.
  - []:([n(if_expression),n(simple_expression),n(code_expression),n(part_eval_function_expression),n(match_expression)])
  vs.
  - []:([n(simple_expression),n([t(if),n(expression),t(then),n(expression)])
  *([t(else),n(expression),t(then),n(expression)])],t(else),n(expression)])]

- Fail (1/1): simple_expression.
  - []:([n(simple_expr),?([t(COLONCOLON),n(simple_expression)])])
  vs.
  - []:([n(simple_expr),?([t(:),n(logical_expression),?([t(:),n(logical_expression),]))])]

- Fail (1/1): logical_expression.
  - []:([n(logical_term),*(t(T,OR)n(logical_term))])
  vs.
  - []:([n(logical_term),*(t(T,OR)n(logical_term))])

- Fail (1/1): logical_term.
  - []:([n(logical_factor),*(t(T,AND)n(logical_factor))])
  vs.
  - []:([n(logical_factor),*(t(T,AND)n(logical_factor))])

- Fail (1/1): logical_factor.
  - []:([t(T,NOT)n(match)])
  vs.
  - []:([t(T,NOT)n(match)])
```


D.2. MODELICA SPECIFICATION VS. OPENMODELICA

101

30  - Fail (1/1): relation.
    - [ ] , [(n[arithmetic_expression]), ?(, ([t(LESS), t(LESSEQ), t(GREATER), t(GREATERT)]]), n[arithmetic_expression]]]
    vs.
    - [ ] , [(n[arithmetic_expression]), ?(, ([n(rel_op), n[arithmetic_expression]])]

35  - Fail (1/1): arithmetic_expression.
    - [ ] , [(n[unary_arithmetic_expression]), *(, ([t(PLUS), t(MINUS), t(PLUS_EW), t(MINUS_EW)])]
    vs.
    - [ ] , [(?n(add_op)), n(term)], *(, ([n(add_op), n(term)])]

40  - Fail (1/1): term.
    - [ ] , [(n(factor)), *(, ([t(STAR), t(SLASH), t(STAR_EW), t(SLASH_EW)])]
    vs.
    - [ ] , [(n(factor)), *(, ([n(mul_op), n(factor)])]

45  - Fail (1/1): factor.
    - [ ] , [(n(primary)), ?(, ([t(POWER), t(POWER_EW)])]
    vs.
    - [ ] , [(n(primary)), ?(, ([t(\`), t(\`)])]

50  - Fail (1/1): primary.
    - [ ] , [(t(UNSIGNED_INTEGER), t(UNSIGNED_REAL), t(STRING), t(TFALSE), t(TTRUE), n(component_reference.function_call), [t(ID), n(function_call)])]
    vs.
    - [ ] , [(t(UNSIGNED_NUMBER), t(STRING), t(TRUE), t(FALSE), t(INITIAL)], n(function_call_args), n(component_reference), [t(\{), n(output_expression_list), t(\})]], t(f
t(component_reference_list), t(\}))], t(function_arguments, t(\}))], t(end))

55  - Fail (1/1): component_reference.
    - [ ] , [(?t(DOT), n(component_reference2)], t(ALLWILD), t(WILD)])
    vs.
    - [ ] , [(t(\), t(IDENT)), n(array_subscripts)], *(, ([t(\), t(IDENT)], n(array_subscripts))]

60  - Fail (1/1): function_arguments.
    - [ ] , [(n(function_argument)], ?(, ([n(\), n(function_arguments)])]
    vs.
    - [ ] , [(n(function_argument)], ?(, ([t(COMMA), n(function_arguments)])]

65  - Fail (1/1): named_arguments.
    - [ ] , [(n(named_argument)], ?(, ([t(COMMA), n(named_arguments)])]
    vs.
    - [ ] , [(t(IDENT), t(=)], n(function_argument)]

70  - Fail (1/1): output_expression_list.
    - [ ] , [(t(RPAR), [t(COMMA), n(output_expression_list)], n(output_expression_list), t(RPAR)])]
    vs.
    - [ ] , [(n(output_expression)], *(, ([t(\), ?(n(output_expression)])]

75  - Fail (1/1): expression_list.
    - [ ] , [(n(expression)], ?(, ([t(COMMA), n(expression_list)])]
    vs.
    - [ ] , [(n(expression)], *(, ([t(\), n(subscript_list)])]

80  - Fail (1/1): array_subscripts.
    - [ ] , [(t(LBRACK), n(subscript_list), t(RBRACK)])
    vs.
    - [ ] , [(t(\)), n(subscript)], *(, ([t(\)), n(subscript)])]
APPENDIX D. COMPARATOR OUTPUT

Listing D.3: Initial Comparator Output of the Modelica Specification vs. OpenModelica

```
Fail (1/1): subscript.
- [] ;( [n(expression), t(COLON)])
vs.
- [] ;( [t(:), n(expression)])
- Ok: comment.

Fail (1/1): string_comment.
- [] ;( [t(STRING), *( [t(PLUS), t(STRING)] )])
vs.
- [] ;( [t(STRING), *( [t(+), t(STRING)] )])

Fail (1/1): annotation.
- [] ;( [t(ANNOTATION), n(class_modification)])
vs.
- [] ;( [t(ANNOTATION), n(class_modification)])
- Roots agree.

Diffing bgf/expr-o.rename.sub.exp.rel.mul.una.ifexp.output.matrix.ref.for.fct.call.part.smp.eliminate.bgf
- Names of defined nonterminals differ.
- Intersection [expression, simple_expression, logical_expression, logical_term, logical_factor, relation, arithmetic_expression, term, factor, primary, component_reference, function_call_args, named_arguments, named_argument, expression_list, array_subscripts, subscript, comment, string_comment, annotation, rel_op, mul_op, add_op, output_expression_list, function_arguments, name, function_argument].
- bgf/expr-o.rename.sub.exp.rel.mul.una.ifexp.output.matrix.ref.for.fct.call.part.smp.eliminate.bgf only: [part_eval_function_expression, for_or_expression_list].
- bgf/expr-m.bgf only: [ ].
- Comparisons per (common) nonterminal:
  - Fail (1/1): expression.
    - [] ;( [t(if), n(expression), t(then), n(expression)], *( [t(else_if), n(expression), t(then), n(expression)]), t(else), n(expression)])
    vs.
    - [] ;( [n(simple_expression), *( [t(if), n(expression), t(then), n(expression)], *( [t(else_if), n(expression), t(then), n(expression)]))], t(else), n(expression)])
    - Ok: simple_expression.
    - Ok: logical_expression.
    - Ok: logical_term.
    - Ok: logical_factor.
    - Ok: relation.
    - Ok: arithmetic_expression.
    - Ok: factor.
    - Fail (1/1): primary.
      - [] ;( [ [n(name), t(der)]], n(function_call_args)], t(UNSIGNED_NUMBER), t(STRING), t(false), t(true), n(component_reference), [t(initial), t(), t()])
      vs.
```
D.3. OPENMODELICA VS. JMODELICA

Listing D.4: Final Comparator Output of the Modelica Specification vs. OpenModelica

D.3 OpenModelica vs. JModelica

- Names of defined nonterminals differ.
  - Intersection: array_subscripts, subscript, annotation, expression, simple_expression, logical_expression, relation, arithmetic_expression, term, factor, primary, output_expression_list, function_call_args, function_arguments, named_arguments, function_argument, named_argument, expression_list, comment,
D.3. OPENMODELICA VS. JMODELICA

Listing D.5: Initial Comparator Output of OpenModelica vs. JModelica

```
vm@vm-VirtualBox:~/Dokumente/impl/lcf$ ../../slps-master/shared/tools/gdt bgf/tgt-j.tgt.bgf bgf/tgt-o.elim.bgf
Normalizing bgf/tgt-j.tgt.bgf.
Normalizing bgf/tgt-o.elim.bgf.
Differing bgf/tgt-j.tgt.bgf and bgf/tgt-o.elim.bgf.
5 - Names defined nonterminals differ.
   - Intersection [array_subscripts, subscript, annotation, expression, simple_expression, logical_expression, relation, arithmetic_expression, term, factor, primary_output_expression_list, function_call_args, function_arguments, named_arguments, function_argument, named_argument, expression_list, comment, component_reference, rel_op, add_op, mul_op, string_comment, logical_factor, part_eval_function_expression, for_or_expression_list].
   - bgf/tgt-j.tgt.bgf only: [].
   - bgf/tgt-o.elim.bgf only: [logical_term].
5 - Comparisons per (common) nonterminal:
   - Ok: array_subscripts.
   - Ok: subscript.
   - Ok: annotation.
   - Fail (1/1): expression.
   - vs.
   - Ok: simple_expression.
```

Appendix D. Comparator Output

- Fail (1/1): logical_expression.
  - [ ] , ( [ n ( logical_factor ) , * ( , ( [ t ( or ) , t ( and ) ] ) , n ( logical_expression ) ) ] )
  vs.
  - [ ] , ( [ n ( logical_term ) , * ( , ( [ t ( or ) , n ( logical_term ) ] ) ) ] )
  Ok: relation.

- Fail (1/1): arithmetic_expression.
  - [ ] , ( [ n ( term ) , * ( , ( [ n ( add_op ) , n ( term ) ] ) ) ] )
  vs.
  - [ ] , ( [ n ( logical_factor ) , * ( , ( [ n ( add_op ) , n ( term ) ] ) ) ] )
  Ok: term.

- Fail (1/1): factor.
  - [ ] , ( [ n ( expression ) , ? ( , ( [ t (ˆ) , t ( . ˆ ) ] ) ) ) ]
  vs.
  - [ ] , ( [ n ( primary ) , ? ( , ( [ t (ˆ) , t ( . ˆ ) ] ) ) ) ]
  Fail (1/1): primary.

- Fail (1/1): function_arguments.
  - [ ] ; ( [ n ( expression ) , t ( for ) , n ( for_indices ) ] )
  vs.
  - [ ] ; ( [ n ( expression ) , t ( for ) , n ( for_indices ) ] )
  Ok: function_arguments.

- Fail (1/1): named_argument.
  - [ ] , ( [ t ( IDENT ) , t ( = ) , n ( function_argument ) ] )
  vs.
  - [ ] , ( [ t ( IDENT ) , t ( = ) , n ( function_argument ) ] )
  Ok: named_argument.

- Fail (1/1): logical_factor.
  - [ ] , ( [ n ( logical_factor ) , * ( , ( [ t ( or ) , t ( and ) ] ) , n ( logical_expression ) ) ] )
  vs.
  - [ ] , ( [ n ( logical_term ) , * ( , ( [ t ( or ) , n ( logical_term ) ] ) ) ] )
  Fail (1/1): logical_expression.

- Fail (1/1): arithmetic_expression.
  - [ ] , ( [ n ( term ) , * ( , ( [ n ( add_op ) , n ( term ) ] ) ) ] )
  vs.
  - [ ] , ( [ n ( logical_factor ) , * ( , ( [ n ( add_op ) , n ( term ) ] ) ) ] )
  Fail (1/1): logical_expression.
D.3. OPENMODELICA VS. JMODELICA

− Ok: `part_eval_function_expression`.
− Fail (1/1): `for_or_expression_list`.
  − `[[], ([n(function_argument), t(for), n(for_indices)])], ([n(function_argument), *([t(., n(function_argument))]), t(., n(named_arguments))]), n(named_arguments)]`
  vs.
  − `[[], ?(n(function_argument), ?([t(., n(function_argument))]), t(., n(named_arguments)))]`
  − Roots agree.

Listing D.6: Final Comparator Output of OpenModelica vs. JModelica
Appendix E

Grammar Transformations

The grammar transformations in this chapter are given in the order they are applied. All transformations independent of whether they lead to convergence of the productions targeted are included. They are pretty-printed versions of the original XML files. Each block is one XBGF file as specified within the corresponding LCF. Apart from the renamings, one XBGF file caters one fail note or one unit of productions modeling a certain construct within the grammar.

E.1 Modelica Specification vs. JModelica

The renamings in this XBGF file dissolve three nominal mismatches, which arise because JModelica uses shorter nonterminal symbols. They are abbreviated versions of the Modelica specification symbols.

```
renameN(exp, expression);
renameN(log_exp, logical_expression);
renameN(aritm_exp, arithmetic_expression);
```

Listing E.1: Renaming of Nonterminals

The transformations in this XBGF file rename terminals. JModelica uses uppercase letters to denote terminals. Operators and parentheses are not specified by the symbol but by the name of the symbol written out. The terminals are transformed into the specification versions. The flex definition of JModelica confirms the translation of the terminals into lexical units that coincide with the Modelica specification.

```
renameT("LBRACK", "[");
renameT("RBRACK", "]");
renameT("LPAREN", "(");
renameT("RPAREN", ")");
renameT("LBRACE", "{");
renameT("RBRACE", "}");
renameT("COLON", ":");
renameT("COMMA", ",");
renameT("EQUALS", ";");
renameT("ANNOTATION", "annotation");
renameT("IF", "if");
renameT("THEN", "then");
renameT("ELSEIF", "elseif");
renameT("ELSE", "else");
renameT("OR", "or");
```
APPENDIX E. GRAMMAR TRANSFORMATIONS

renameT("AND", "and");
renameT("NOT", "not");
renameT("TRUE", "true");
renameT("FALSE", "false");
renameT("FOR", "for");
renameT("END", "end");

Listing E.2: Renaming of Terminals

The transformations alter the JModelica productions subscript_list and subscript to converge with the Modelica specification. The transformations alter the productions in the following way:

```plaintext
subscript_list : subscript | subscript_list "," subscript
```

```plaintext
array_subscripts : ["" subscript_list "]
```

```plaintext
vertical(in subscript_list);
deyaccify(subscript_list);
inline(subscript_list);
```

Listing E.3: Refactoring of Subscripts

The XBGF transformations for the if expressions alter the production elseif_exp in a similar way as described for subscripts. Inlining completes the convergence.

```plaintext
vertical(in elseif_exp);
deyaccify(elseif_exp);
inline(elseif_exp);
inline(if_exp);
```

Listing E.4: Refactoring of If Expressions

To align the simple expressions, we perform the same two transformations twice. First, we factor out a logical expression and massage the epsilon choice into an optional. Second, we factor out a colon and a logical expression and massage the epsilon choice again. The states of the definition of the nonterminal simple_expression are, with l_e standing for logical_expression:

```plaintext
l_e | l_e ":" l_e | l_e ":" l_e ":" l_e
```

```plaintext
factor((logical_expression | (logical_expression ":" logical_expression) | (logical_expression ":" logical_expression ":" logical_expression)),
logical_expression (EPSILON | (":" logical_expression) | (":" logical_expression ":" logical_expression)));
massage((EPSILON | (":" logical_expression) | (":" logical_expression ":" logical_expression) | (":" logical_expression ":" logical_expression) | (":" logical_expression ":" logical_expression)),
factor(((":" logical_expression) | (":" logical_expression ":" logical_expression) | (":" logical_expression ":" logical_expression) | (":" logical_expression ":" logical_expression) | (":" logical_expression ":" logical_expression)),
massage((EPSILON | (":" logical_expression)),
```
E.1. MODELICA SPECIFICATION VS. JMODELICA

Next, we consider the relations. With the different relational operators between two arithmetic expressions in an own choice, we need to factor out the front and end arithmetic expression and extract the operators. Then, we move on to align the rest of the production arithmetic_expression by factoring and massaging in the same way as we have done before with the simple expressions.

```plaintext
factor( 
    (arithmetic_expression "LT" arithmetic_expression) | (arithmetic_expression "LEQ" arithmetic_expression) | (arithmetic_expression "GEQ" arithmetic_expression) | (arithmetic_expression "EQ" arithmetic_expression) | (arithmetic_expression "GT" arithmetic_expression) | (arithmetic_expression "NEQ" arithmetic_expression) | arithmetic_expression) , 
    (arithmetic_expression) ;

renameT("LT", "<");
renameT("LEQ", "\leq");
renameT("GEQ", "\geq");
renameT("EQ", "=");
renameT("GT", ">");
renameT("NEQ", "\neq");
extract( rel_op : "<" | "\leq" | "\geq" | "=" | "\neq" | ">");

factor( 
    (arithmetic_expression rel_op arithmetic_expression) | arithmetic_expression) , 
    arithmetic_expression (EPSILON));

massage( 
    (rel_op arithmetic_expression) | EPSILON), 
    (rel_op arithmetic_expression) ?);
```

The transformations in this block concern the arithmetic operators which we need to extract, as well. To be able to do that, we need to factor out the terms in front and behind the operators and extract them.

```plaintext
factor( 
    (multiplicative_exp | (additive_exp "PLUS" multiplicative_exp) | (additive_exp "MINUS" multiplicative_exp) | (additive_exp "DOTPLUS" multiplicative_exp) | (additive_exp "DOTMINUS" multiplicative_exp)) , 
    (multiplicative_exp | (additive_exp (("PLUS" multiplicative_exp) | ("MINUS" multiplicative_exp) | ("DOTPLUS" multiplicative_exp) | ("DOTMINUS" multiplicative_exp)))));

factor( 
    (multiplicative_exp | (additive_exp (("PLUS" multiplicative_exp) | ("MINUS" multiplicative_exp) | ("DOTPLUS" multiplicative_exp) | ("DOTMINUS" multiplicative_exp))));
```
APPENDIX E. GRAMMAR TRANSFORMATIONS

Listing E.7: Refactoring of Arithmetic Operators

The terminals that denote a power symbol need not be extracted. They have to be in the same form of a choice but not in an own production, rather as an inner choice. Thus, we factor out the start and end terms but do not extract the inner choice. We finish the transformation chain with massaging the epsilon choice and renaming the terminals and the production head. The steps in the transformation are the following; \( p \) stands for primary and the power symbols are already renamed for this description:

\[

d + n | - n | .+ | .- \\

\]

Listing E.8: Refactoring of Power Expressions

The arithmetic expression is solely defined by an additive expression that is not part of the Modelica specification. Thus, we inline the deyaccified additive expression. The multiplicative
expression is also deyaccified and in addition, renamed. At this point, JModelica includes a unary expression that holds the sign symbols. It means that the production `term` fails while the production `factor` agrees and we have an exclusive production for JModelica. Later, we will dissolve the exclusive production which moves the signs into the production `factor`. The result is that we have one exclusive production less, the terms agree, and the factors fail.

```plaintext
vertical( in additive_exp );
deyaccify( additive_exp );
inline( additive_exp );
vertical( in multiplicative_exp );
deyaccify( multiplicative_exp );
renameN( multiplicative_exp, term );
```

Listing E.9: Refactoring of Arithmetic Expressions

The states the comments pass through during transformation are:

```plaintext
string_comment: "STRING" | string_comment "+" "STRING"
=> string_comment: "STRING" ( "+" "STRING" )
comment: string_comment ? annotation ?
=> comment: ( "STRING" ( "+" "STRING" )* )? annotation ?
string_comment: ( "STRING" ( "+" "STRING" )* )?
```

Listing E.10: Refactoring of Comments

We introduce a new nonterminal, logical factors, by extracting the respective part of the logical expressions. The commonly used combination of factoring and massaging completes the convergence.

```plaintext
extract( logical_factor:
    "not" relation | relation
);
factor( ("not" relation) | relation ),
("not" | EPSILON) relation);
massage( ("not" | EPSILON),
"not" ?);
```

Listing E.11: Refactoring of Logical Factors

The matrices in JModelica model one choice in the primaries and the expression lists in the Modelica specification. The transformation `deyaccify` resolves the differences together with renamings and inlining.

```plaintext
vertical( in matrix_row );
deyaccify( matrix_row );
renameN( matrix_row, expression_list );
vertical( in matrix );
deyaccify( matrix );
```
Listing E.12: Refactoring of Matrices

The following transformations deal with aligning the component references. The first three transformations do the following:

```
parsesAccessLoc = parsesAccessSingle | parsesAccessSingle DOT
```

The inline transformations eliminate nominal mismatches, the renamings align nonterminal and terminal symbols. The first line of the following listing depicts the definition of `parsesAccess` after distribution. The other lines show the result of the transformations, excluding the last unfolding.

```
parsesAccess : "IDENT" array_subscripts? ( DOT parsesAccess array_subscripts? ) |
| DOT "IDENT" |
| DOT "IDENT" DOT array_subscripts? DOT "IDENT" DOT array_subscripts? |

⇒ parsesAccess : "IDENT" array_subscripts? ( DOT parsesAccess array_subscripts? ) |
| DOT "IDENT" DOT ( DOT parsesAccess array_subscripts? DOT "IDENT" DOT array_subscripts? ) |

⇒ parsesAccess : "IDENT" array_subscripts? ( helper ) |
| DOT "IDENT" DOT ( DOT parsesAccess array_subscripts? ) |

⇒ parsesAccess : "IDENT" array_subscripts? ( helper ) |
| DOT "IDENT" DOT ( DOT parsesAccess array_subscripts? ) |

⇒ parsesAccess : "IDENT" array_subscripts? ( helper ) |
| DOT "IDENT" DOT ( DOT ParsesAccess array_subscripts? ) |

⇒ parsesAccess : "IDENT" array_subscripts? ( helper ) |
| DOT "IDENT" DOT ( DOT parsesAccess array_subscripts? ) |
```

The last transformation unfolds this definition into the definition of component references, which has been defined solely by the nonterminal `parsesAccess`. The parse access expression becomes the names expression later on.

```
vertical ( in parsesAccessLoc );
deyaccify ( parsesAccessLoc );
massage ( |
| parsesAccessSingle DOT parsesAccessSingle , |
| parsesAccessSingle DOT parsesAccessSingle* ) ; |
| parsesAccessSingle DOT parsesAccessSingle ; |
| parsesAccessSingle DOT parsesAccessSingle ; |
| inlinex ( parsesAccessSingle ) ; |
| inlinex ( firstClassAccess ) ; |
| renameN ( accessExpression , componentReference ) ; |
| renameT ( "DOT" , ")" ) ; |
| distribute ( in parsesAccess ) ; |
| factor ( |
| "IDENT" array_subscripts? DOT ( DOT "IDENT" array_subscripts? ) DOT ( DOT "IDENT" array_subscripts? ) |
| ( DOT "IDENT" DOT array_subscripts? DOT "IDENT" DOT array_subscripts? ) ) |
| ( DOT "IDENT" DOT array_subscripts? DOT ( DOT EPSILON DOT "IDENT" DOT array_subscripts? ) ) |
| ( DOT "IDENT" DOT array_subscripts? DOT ( DOT "IDENT" DOT array_subscripts? ) ) ) ) ; |
| massage ( |
The states the production named_arguments passes through are:

\[
\begin{align*}
\text{named_arguments} & : \text{named_argument} | \text{named_arguments} "," \text{named_argument} \\
\Rightarrow & \quad \text{named_arguments} : \text{named_argument} ("," \text{named_argument}) \\
\Rightarrow & \quad \text{named_arguments} : \text{named_argument} ("",")* \text{named_argument} \\
\Rightarrow & \quad \text{named_arguments} : \text{named_argument} | \text{named_argument} "," \text{named_arguments} \\
\Rightarrow & \quad \text{named_arguments} : \text{named_argument} (< | "," \text{named_arguments}) \\
\end{align*}
\]

The last two transformations align the production named_argument.

We deal with partial function calls. We inline the corresponding production to eliminate the exclusive nonterminal and align the production function_argument. The nonterminal parse_access becomes name, the terminal "FUNCTION" becomes "function".

\[
\begin{align*}
\text{inl ine} (\text{partial_function_call})
\Rightarrow & \quad \text{renameT} ("FUNCTION", "function") \\
\Rightarrow & \quad \text{renameN} (\text{parse_access}, \text{name}) \\
\end{align*}
\]
We introduce the formerly exclusive Modelica specification nonterminal function_call_args by extracting the definition out of the production function_call. Then, we inline the function calls to eliminate the exclusive OpenModelica nonterminal function_call and to align the structure of the primary definition. The next transformations aligns the production function_arguments. The last two transformations inline the “der” expressions into the primaries and rename a terminal.

extract(
  function_call_args:
  "(" function_arguments? ")"
in function_call);
inline(function_call);
vertical(in arg_list_p);
deyaccify(arg_list_p);
inline(dér_expression);
renameT("DER" , "der");

Listing E.16: Refactoring of Function Calls and Function Arguments

The transformations eliminate two choices of the production primary.

vertical(in primary);
removeV(
  primary:
  "UNSIGNEDINTEGER"
);
removeV(
  primary:
  "TIME"
);
horizontal(in primary);

Listing E.17: Elimination of Redundant Choices

The final refactorings transform the logical expressions, eliminate the unary expressions, and introduce output expression lists. The first block alters the definition of logical expressions, abbreviated by l_e, logical factors are abbreviated by l_f:

\[
\begin{align*}
\text{l_e} & \text{ or } \text{l_e} \mid \text{l_e} \text{ and } \text{l_e} \mid \text{l_f} \\
\Rightarrow & \text{l_e} ( \text{ or } \text{l_e} \mid \text{l_e} \text{ and } \text{l_e} ) \mid \text{l_f} \\
\Rightarrow & \text{l_e} ( \text{ or } \text{l_e} \text{ and } ) \mid \text{l_e} \mid \text{l_f} \\
\Rightarrow & \text{l_f} ( ( \text{ or } \text{l_e} \text{ and } ) \mid \text{l_e} ) \\
\end{align*}
\]

The unary expressions change through the course of transformation in the following way:

\[
\begin{align*}
\text{unary_exp} & : \text{factor} \mid \text{"-" unary_exp} \mid \text{"+" unary_exp} \\
\Rightarrow & \text{unary_exp} : \text{factor} \mid ( \text{"-"} \text{"+"} ) \text{ unary_exp} \\
\Rightarrow & \text{unary_exp} : ( \text{"-" \"+"} ) \text{ primary } ( ( \text{"-" \"+"} ) \text{ primary } )? \\
\Rightarrow & \text{factor} : ( \text{"-" \"+"} ) \text{ primary } ( ( \text{"-" \"+"} ) \text{ primary } )?
\end{align*}
\]

To extract the output expression list which is in JModelica a single expression, we have to temporarily extract the “der” expression to be able to extract only the intended expression in the definition of the primaries. The last three transformations handle the introduction of output expression lists.
E.2 Modelica Specification vs. OpenModelica

Due to the higher alignment in structure, OpenModelica and the Modelica specification share more nonterminals than the specification and JModelica. Renamings of nonterminals occur later in the process when working through the fail notes but not in the beginning like in the LCI of the Modelica specification and JModelica. The process starts with renamings of terminals.

```plaintext
- renameT("T_OR","or");
- renameT("T_AND","and");
- renameT("T_NOT","not");
- renameT("LESS","<");
- renameT("LESSEQ","<=");
- renameT("GREATER">");
- renameT("GREATEREQ",">=");
- renameT("EQEQ","==");
- renameT("LESSGT","<>" AUTHOR: E.2d Modelica Specification vs. OpenModelica

Listing E.18: Final Refactorings for Clarification of Differences

logical_expression ("or" logical_expression) | ("and" logical_expression));
factor( (("or" logical_expression) | ("and" logical_expression)),
  ("or" | "and") logical_expression);
extract( op:
  "or" | "and"
);
vertical( in logical_expression );
deacify(logical_expression);
inline (op);
factor( (("-" unary_exp) | ("+" unary_exp)),
  ("-" | "+") unary_exp);
extract( op:
  "+" | "+"
);
vertical( in unary_exp );
deacify(unary_exp);
inline (op);
inline (factor);
renameN(unary_exp, factor);
extract( der_prod:
  "der" "(" expression ")"
);
extract( output_expression_list:
  expression
  in primary);
inline (der_prod);
```
Appendix E. Grammar Transformations

renameT ("POWER", "^" );
renameT ("POWERW", "\^" );
renameT ("T.FALSE", "false" );
renameT ("T.TRUE", "true" );
renameT ("DER", "der" );
renameT ("LPAR", "(" );
renameT ("RPAR", ")" );
renameT ("LBRACK", "[ "");
renameT ("RBRACK", "] "");
renameT ("LBRACE", "{ "");
renameT ("RBRACE", "} "");
renameT ("T.END", "end" );
renameT ("DOT", ". "");
renameT ("COMMA", "," );
renameT ("EQUALS", ":=" );
renameT ("COLON", "::" );
renameT ("FUNCTION", "function" );
renameT ("OPERATOR", "operator" );
renameT ("AS", "as" );
renameT ("COLUMN", "::" );

Listing E.19: Renaming of Terminals

The subscript lists need to be inlined. Before doing so, we need to transform the definition into the form subscript (" ", subscript )\* . The steps towards convergence lead to the following interim definitions of subscript lists:

```
subscript (" ", subscript_list )?
  => subscript ( \epsilon | " ", subscript_list )
  => subscript | subscript " ", subscript_list
  => ( subscript " ", )* subscript
  => subscript (" ", subscript )*
```

```
massage(
  (" ", subscript_list )?,
  {(" ", subscript_list ) | EPSILON});
vertical( in subscript_list );
deyaccify ( subscript_list );
massage(
  (subscript " ")* subscript,
  subscript (" ", subscript )*)
inline ( subscript_list );
```

Listing E.20: Refactoring of Subscripts

Expression lists are transformed in the same way as are subscript lists, leading to the following transformation process:

```
expression (" ", expression_list )?
  => expression ( \epsilon | " ", expression_list )
  => expression | expression " ", expression_list
  => ( expression " ", )* expression
  => expression (" ", expression )*
```

```
massage(
  (" ", expression_list )?,
  {(" ", expression_list ) | EPSILON});
distribute( in expression_list );
vertical( in expression_list );
```
E.2. MODELICA SPECIFICATION VS. OPENMODELICA

deyaccify(expression_list);
massage((expression "","" expression,
expression ("","" expression)*));

Listing E.21: Refactoring of Expression Lists

The relational, additive, and multiplicative operators already are in an inner choice that can be extracted. The next two transformation listings do exactly that for relational and multiplicative operators.

extract(
  rel_op: "<" | "<=" | "=" | ">=" | ">=" | "<>"
);

Listing E.22: Refactoring of Relational Operators

extract(
  mul_op: "*" | "/" | ".*" | "./"
);

Listing E.23: Refactoring of Multiplicative Operators

For the additive operators, we first need to bring the additive operators in the definition of unary expressions into the form of an inner choice. The extraction of the operators replaces every occurrence of them. This way, they already appear in the extractable form in the unary expression when extracted and we do not have to fold the definition of the operators into the production afterwards. The refactoring of the inner choice of additive operators in the unary expressions generates an epsilon alternative which is massaged in the end into an optional. The inlining of the unary expressions aligns the arithmetic expressions and eliminates the exclusive OpenModelica nonterminal unary_arithmetic_expression.

factor(
  (("+" term) | ("-" term) | (".+" term) | (".-" term) | term),
  ("+" | "-" | ".+" | ".-" | EPSILON) term);

Listing E.24: Refactoring of Unary Expressions

If expressions need a few inlinings, renamings and a refactoring of the “elseif ” part. Elseif expression lists are transformed in the same way as subscript and expression lists, except that the last step includes a different massaging operation.

elseif_expression_list : elseif_expression elseif_expression_list?
=> elseif_expression_list : elseif_expression ( ε | elseif_expression_list )
=> elseif_expression_list : elseif_expression elseif_expression_list
elseif_expression_list

epsilon

(ε | elseif_expression_list )
APPENDIX E. GRAMMAR TRANSFORMATIONS

Listing E.25: Refactoring of If Expressions

For the output expression lists, we need transformations that move the closing parenthesis to the outer right and that converge the remainder of the definitions. The recursion must be transformed into an iteration.

```
=> elseif_expression_list : ( elseif_expression )* elseif_expression
5 => elseif_expression_list : ( elseif_expression )+
    if_expression : "IF" expression "THEN" expression ( ( elseif_expression )+ )? "ELSE"
    elseif_expression : "IF" expression "THEN" expression ( elseif_expression )* "ELSE"
    elseif_expression : "if" expression "then" expression ( elseif_expression )* "else"
    elseif_expression : "if" expression "then" expression ( elseif_expression )* "else"

massage(
    elseif_expression_list?,
    ( elseif_expression_list | EPSILON)
    in elseif_expression_list);
5 distribute( in elseif_expression_list );
vertical( in elseif_expression_list );
deyaccify( elseif_expression_list );
inline( elseif_expression_list );
massage(
    elseif_expression+?,
    elseif_expression*)
renameT("IF", "if");
renameT("THEN", "then");
renameT("ELSEIF", "else");
renameT("ELSE", "else");
inline( elseif_expression );
inline( if_expression );
```

```
Listing E.25: Refactoring of If Expressions

For the output expression lists, we need transformations that move the closing parenthesis to the outer right and that converge the remainder of the definitions. The recursion must be transformed into an iteration.

```
")" | "," output_expression_list |
    expression ( "," output_expression_list |
    expression ")"
=> ")" | ( ( expression ) "," output_expression_list |
    expression ")"
=> ( ( expression ) "," output_expression_list |
    expression ")"
5 => expression? ")" | expression? "," output_expression_list
=> ( expression? "," expression? ")")
=> ( expression? "," expression? // inlined and extracted again without ")"
=> expression? ( "," expression? ")"

distribute( in output_expression_list );
factor(
    ("", output_expression_list) |
    (output_expression_list | EPSILON | expression )"," output_expression_list);
5 factor(
    ("", output_expression_list | EPSILON | expression )""),
    (EPSILON | expression )""
    massage(
        (EPSILON | expression),
        expression?)
    vertical( in output_expression_list );
deyaccify( output_expression_list );
inline( output_expression_list );
extract( output_expression_list );
```
Matrix expression lists make up one choice of the production primary. The transformations needed form a common pattern.

```
expression_list ( "SEMICOLON" matrix_expression_list)? => expression_list ( ε | "SEMICOLON" matrix_expression_list ) => ( expression_list "SEMICOLON" )* expression_list
5 => expression_list ( ":;" expression_list )* // transformations renameT and massage
```

```
massage(( "SEMICOLON" matrix_expression_list)? , (EPSILON | ("SEMICOLON" matrix_expression_list)));
distribute( in matrix_expression_list );
5 vertical( in matrix_expression_list );
deyaccify(matrix_expression_list);
renameT("SEMICOLON", ";");
massage(( expression_list ";;" )* expression_list ,
expression_list ( ";;" expression_list)*);
inline(matrix_expression_list);
```

The production component_reference2 accounts for the main definition of component references we need to align with Modelica. The transformations lead to the following chain of definitions of component references of type 2:

```
("IDENT" | "OPERATOR") array_subscripts? ( "." component_reference2 )?
=> helper ( ε | "." component_reference2 ) // to not distribute the first choice
=> helper | helper "." component_reference2
5 => ( helper "." )* helper
=> helper ( "." helper*)
=> ("IDENT" | "OPERATOR") array_subscripts? ( "." ("IDENT" | "OPERATOR")
array_subscripts? )*
```

```
massage(( "IDENT" | "operator") array_subscripts? ,
(EPSILON | ("." component_reference2)));
5 extract( helper:
("IDENT" | "operator") array_subscripts? );
distribute( in component_reference2 );
vertical( in component_reference2 );
deyaccify(component_reference2);
massage(
  (helper ".")* helper,
  helper ("." helper)*);
inline(helper);
10 inline(component_reference2);
```
The for-or-expression lists, abbreviated \texttt{for\_or}, change in the following way:

\begin{verbatim}
for\_or2: \epsilon | expression ("," for\_or2 )?
\Rightarrow for\_or2: \epsilon | expression ( \epsilon | "", for\_or2 )
\Rightarrow for\_or2: expression? | expression ",", for\_or2
\Rightarrow for\_or2: ( expression ",", expression? 
for\_or: \epsilon | expression ( ",", ( expression ",", expression? | "FOR" for\_indices )
\Rightarrow for\_or: (expression (",", ( expression ",")* expression? | "FOR" for\_indices ) )?

massage(
("", for\_expression\_list2)?,
(EPSILON | ",", for\_expression\_list2)));
distribute( in for\_expression\_list2 );
5 massage(
(EPSILON | expression).
expression?);
vertical( in for\_expression\_list2 );
deyaccify(for\_expression\_list2);
10 inline(for\_expression\_list2);
massage(
(EPSILON | expression (""," ( expression ",")* expression? | "FOR" for\_indices ) )?),
(expression ("", ( expression ",")* expression? | "FOR" for\_indices )?);
renameT("FOR"," for\_or_indices");
\end{verbatim}

Listing E.29: Refactoring of For-or-expression Lists

The upcoming XBGF file deals with function arguments. OpenModelica defines the function arguments through for-or-expression lists we have transformed in the previous listing. We will unfold the definition of for-or-expression lists into the function arguments as the Modelica specification does not know the limited form for-or-expression list for function arguments. The transformations form the following chain of definitions of function arguments:

\begin{verbatim}
for\_expression\_list named\_arguments?
\Rightarrow ( expression ("," ( expression ",")* expression? | "for" for\_indices ) )?
\Rightarrow helper? ( \epsilon | named\_arguments )
\Rightarrow ( \epsilon | helper ) ( \epsilon | named\_arguments )
5 \Rightarrow \epsilon | helper | named\_arguments | helper named\_arguments
\Rightarrow ( helper | named\_arguments | helper named\_arguments )?
\Rightarrow helper | named\_arguments | helper named\_arguments // inline and extract without optional
\Rightarrow helper ( \epsilon | named\_arguments ) | named\_arguments
\Rightarrow helper named\_arguments? | named\_arguments
10 \Rightarrow expression (",", ( expression ",")* expression? | "for" for\_indices )
named\_arguments? | named\_arguments

massage(
named\_arguments?,
(EPSILON | named\_arguments)
in function\_arguments);
5 unfold(for\_expression\_list in function\_arguments);
extract(
helper:
expression (",", ( expression ",")* expression? | "FOR" for\_indices )?
);
10 massage(
helper?,
\end{verbatim}
Listing E.30: Refactoring of Function Arguments

The following transformations deal with a production that is not a production of its own in the Modelica specification but part of the production primary, namely component_reference__function_call. The idea behind the transformations are listed next:

\[
\begin{align*}
\text{component_reference function_call_args?} & \mid \text{"INITIAL" "(" ")"} \\
\Rightarrow & \text{component_reference ( \(\epsilon\) | function_call_args) | "INITIAL" "(" ")"} \\
\Rightarrow & \text{component_reference | component_reference function_call_args | "INITIAL" "(" ")"} \\
\Rightarrow & \text{component_reference | name function_call_args | "INITIAL" "(" ")"}
\end{align*}
\]

The last transformation factors the function call arguments of the function calls beginning with a name and the keyword “der” to align the structure of the primary definitions.

Listing E.31: Refactoring of Function Call Arguments
The nonterminal \texttt{function_argument} is not necessary in OpenModelica in the way it is used in Modelica. But to align the productions, we introduce such a nonterminal and define it solely by an expression. The new nonterminal is then folded into several productions where the Modelica specification uses the nonterminal.

\begin{verbatim}
introduce{
  function_argument: expression
};
fold(function_argument in named_argument);
fold(function_argument in for_or_expression_list);
fold(function_argument in function_arguments);
\end{verbatim}

Listing E.32: Refactoring of Function Expressions

Simple expressions can be combined in OpenModelica to a larger extent. However, the core production of a simple expression agrees. Therefore, we shift the additional combinatorial possibilities into the definition of the nonterminal \texttt{expression} and let the definition of the simple expressions be simply the generation of a simple expression. To inline the simple expressions, we need to eliminate the recursion.

\begin{verbatim}
simple_expr ( "::" simple_expression )? | "IDENT" "as" simple_expression
⇒ simple_expr ( ε | "::" simple_expression ) | "IDENT" "as" simple_expression
⇒ simple_expr | simple_expr "::" simple_expression | "IDENT" "as" simple_expression
⇒ simple_expr | ( simple_expr "::" | "IDENT" "as" ) simple_expression
⇒ ( simple_expr "::" | "IDENT" "as" )" simple_expr
\end{verbatim}

Listing E.33: Refactoring of Simple Expressions

We can eliminate two redundant choices within the definition of component references according to the lexer definition.

\begin{verbatim}
vertical( in primary );
removeV( primary:
  "UNSIGNED_INTEGER"
); horizontal( in primary );
renameT("UNSIGNED_REAL", "UNSIGNED_NUMBER");
vertical( in component_reference );
removeV( component_reference:
  "WILD"
); removeV( component_reference:
  "ALLWILD"
\end{verbatim}
E.3 OpenModelica vs. JModelica

The first two XBGF files eliminate the production `name` in each grammar as it is defined in the same way as component references and was only introduced because of Modelica. In neither grammar, names were incorporated in the grammar as an explicit production.

```plaintext
equate(name, component_reference);
```

Listing E.35: Elimination of Names in OpenModelica

```plaintext
equate(name, component_reference);
```

Listing E.36: Elimination of Names in JModelica

The first two transformations partly revert an inlining we applied to JModelica to align the production `function_argument`. We extract the function expression again and fold the definition of function call arguments into it to converge the production `part_eval_function_expression`. The last three transformations introduce the notion of for-or-expression lists though they are defined only by function arguments. It eliminates an exclusive Modelica nonterminal and aligns the definition of primaries more.

```plaintext
extract(
  part_eval_function_expression:
    "function" component_reference "(" function_arguments? ")"
);
fold(function_call_args);
introduce(
  for_or_expression_list:
    function_arguments
);
fold(for_or_expression_list in primary);
unfold(function_arguments in for_or_expression_list);
```

Listing E.37: Refactoring of JModelica
Bibliography


