

FRIEDRICH-ALEXANDER-UNIVERSITÄT ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT • DEPARTMENT INFORMATIK

Lehrstuhl für Informatik 10 (Systemsimulation)



Generating IDE Support for Multiple Domain Specific Languages

Michael Welscher

Bachelor Thesis

Generating IDE Support for Multiple Domain Specific Languages

Michael Welscher

Bachelor Thesis

Aufgabensteller: PD Dr.-Ing. habil. Harald Köstler

Betreuer: Sebastian Kuckuk, M.Sc.
Christian Schmitt, M.Sc.

Bearbeitungszeitraum: 2014/11/15 to 2015/04/13

Erklärung:

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch den Lehrstuhl für Systemsimulation (Informatik 10), wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Bachelor Thesis einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 2015/04/13

.....

Abstract (de)

Mit der zunehmenden Weiterverbreitung von Domänenspezifischen Sprachen in vielen Bereichen von Computeranwendungen wird es immer wichtiger, diese Sprachen mit geeigneten integrierten Entwicklungsumgebungen zu unterstützen. Obwohl es zahlreiche Werkzeuge gibt, die die gemeinsame Erschaffung von Sprache und Entwicklungsumgebung unterstützen, ist unklar, wie gut diese Werkzeuge funktionieren wenn die Sprache extern implementiert und entwickelt wird.

Das Ziel dieser Arbeit ist es, verschiedene Language Workbenches und andere, ähnliche Werkzeuge bezüglich ihrer Eignung für einen solchen Ansatz zu evaluieren und dies mit der Implementierung einer Fallstudie in einem der geeignet erscheinenden Werkzeuge zu belegen. Um den Evaluationsprozess vorzubereiten erfolgt eine ausführliche Erklärung zu allgemeinen Fähigkeiten von integrierten Entwicklungsumgebungen, insbesondere im Hinblick auf Editor Features. Zusätzlich wird die Sprache die für die Fallstudie verwendet wird, ExaSlang aus dem ExaStencils Projekt, in einem Überblick allgemein erläutert. Der Evaluationsprozess wird mit besonderer Wertlegung auf die Anforderungsdefinitionen beschrieben. Die drei am besten passenden Workbenches, wurden auf einer praktischen Ebene getestet und in großem Umfang eingeführt, bevor die finale Wahl für die Fallstudie erläutert wird. Letzendlich wird die Implementierung ausführlich beschrieben und Schlussfolgerungen über die Allgemeingültigkeit der Ergebnisse gezogen.

Abstract (en)

The advent of Domain Specific Languages in many parts of computer applications makes it increasingly important to support those languages with proper Integrated Development Environments. While there are plenty of tools that support the creation of both language and development environment, it is fairly unknown how well those tools work with a language that is being implemented and developed externally.

The goal of the thesis is to evaluate different Language Workbenches and similar tools towards their ability to enable such an approach and proof of the concept, by implementing a case study in one of the, seemingly best fitting, tools. To prepare for the evaluation process, an elaborate explanation of Integrated Development Environment abilities, especially with regard to editor features is given. Additionally an overview of the language used for the case study, ExaSlang of the ExaStencils project is provided. The evaluation process is described with a strong emphasis on the requirement definition. The three best suited workbenches have been tested on a practical base and are introduced in length before the final choice for the case study implementation is exemplified. Last, but not least, the implementation is described thoroughly and conclusions about the generality of the results are made.

Contents

1	Introduction	1
1.1	Overview	1
1.2	Motivation	1
1.3	Outline	2
2	ExaStencil and ExaSlang	3
2.1	ExaStencils Overview	3
2.2	ExaSlang Overview	4
3	Integrated Development Environments (IDEs) and IDE Tools	6
3.1	Purpose and Capabilities of IDEs	6
3.2	Editor Support	7
3.2.1	Syntax Highlighting	7
3.2.2	Code Structuring and Bracket Matching	8
3.2.3	Syntactical Error Detection	9
3.2.4	Syntactic Completion and Templates	10
3.2.5	References, Name Resolving and Type Analyzing	11
3.2.6	Dynamic Tooltips and Variable Templates	11
3.2.7	Outline Generation	12
3.3	Other IDE Features	14
3.3.1	Project Templates	14
3.3.2	Code Compilation and Generation Support	14
3.3.3	Interface and Workbench Adjustments	15
4	Evaluation of LWBs and similar Tools regarding IDE support	16
4.1	Requirements	17
4.1.1	General Requirements	17
4.1.2	Editor Requirements	18
4.1.3	Other Requirements	18
4.2	Evaluation Process	19
4.2.1	First and Second Evaluation Steps	19
4.2.2	Final Evaluation Step	20
4.2.3	Results	27
4.3	Remarks on the Evaluation Process	28
5	The Spoofax Language Workbench	30
5.1	The Syntax Definition Formalism v.3 (SDF3)	31

5.2	The Name Binding Language (NaBL)	34
5.3	The Type Specification Language (TS)	34
5.4	The Spoofox Testing Language (SPT)	35
5.5	The Editor Services Language (ESV)	36
5.6	The Stratego Transformation Language	39
6	ExaSlang in Spoofox and Eclipse	41
6.1	Spoofox Features	41
6.1.1	Syntax Highlighting	42
6.1.2	Outliner	43
6.1.3	Completions	47
6.1.4	Reference and Name Resolving	49
6.1.5	Folding	52
6.2	Eclipse Features	52
6.2.1	Project Templates	54
6.2.2	UI Extension	56
6.2.3	Plugin Deployment and Update Site	58
6.3	Other Features	60
6.3.1	Language Project Husks	60
6.3.2	Spoofox Runtime Adjustments	61
7	Overall Results	62
7.1	Implementation Time to Endresult	62
7.2	Recommendations for Future Projects	62
7.3	Feasibility for other Languages	63
8	Future Work	64
8.1	Possible Features for Future Versions	64
8.2	Automation of Domain Specific Language (DSL) Transfer and Feature creation	64
8.3	Implementing the other ExaSlang Languages	65
9	Conclusion	66
	Bibliography	a
	List of Figures	c
	List of Tables	e
	List of Snippets	f
	Acronyms	h
A	ExaSlang IDE Installation Instructions	i

B	ExaSlang License: GNU Lesser Public License (GLPL)	m
C	Survey: Time Saved by IDEs	q

1 Introduction

1.1 Overview

The aim of this thesis is to show that already specified and implemented Domain Specific Language can be extended with an Integrated Development Environment (IDE) through the help of existing tools. Therefore it describes the implementation of an IDE for the ExaStencil Project's DSLs ExaSlang.

Domain Specific Languages (DSLs)

DSLs are a class of programming languages and stand in contrast to general-purpose languages. While the latter are designed to appeal to as many fields of application as possible, by only defining general constructs, DSLs are created to be used within one specific domain of application. Therefore they incorporate expressions and constructs that reflect specific characteristics of this domain. This is done primarily to ease working in those domains. Depending on the domain, a DSL may be either textual based, or graphical based. The majority of existing DSLs currently are text based [1]. An example for such a language is \TeX ¹, which is designed for the domain of typesetting.

1.2 Motivation

Over the past decades, the increased availability of computing power for scientific calculations and simulations, paired with new approaches to make use of this development, has shown the need to provide tools to allow the proper application of new methods and discoveries [2].

These tools should not be limited to only provide domain specific functionality, like a library of mathematical functions. Moreover they should also provide features that are helpful to the user in a more general way. A proper Graphical User Interface (GUI) can be such a feature. When working with computers, it is often the easiest way to present features and functionality to the end user in a convenient way. If given the choice, most users will choose an application with a GUI over one without. The success of GUIs in general over console based applications at almost all computer workplaces shows this impressively.

The ExaStencil² project aims at developing such tools and providing them to a wide variety of users. It is currently being developed at the Chair of System Simulation (LSS)³

¹<http://www.tug.org>

²<http://www.exastencils.org/>

³<https://www10.informatik.uni-erlangen.de>

and the Chair of Hardware Software Co Design (CoD)⁴ at the Friedrich-Alexander University Erlangen-Nürnberg (FAU)⁵. The development also takes place at the Chair of Programming and the Chair of Software Product Lines from the University of Passau⁶ as well as the Applied Computer Science Group from the University of Wuppertal⁷. As part of ExaStencils, ExaSlang is a set of DSLs. ExaSlang will allow engineers as well as mathematicians, computer scientists and physicists to work on simulation problems in their accustomed environment. They will be enabled to make use of the knowledge of the other domains without having to learn too much outside of their own domain [3]. To provide an easy-to-learn setting for the use of ExaSlang, different tools will be offered to the end user. One of these tools is, of course, an IDE in form of a language specific editor, documentation and a GUI for the language. Examples for such editors are Microsoft's Visual Studio⁸ and Notepad++⁹ for Windows or the platform independent Eclipse Environment¹⁰ that each support many different programming languages.

1.3 Outline

Directly after this section, in chapter 2, a brief overview of ExaStencils and ExaSlang is given. A general description of Integrated Development Environments and IDE features is given in chapter 3, followed by the illustration of the evaluation process and its results in chapter 4 of the thesis.

Spoofax¹¹, the resulting Language Workbench from the evaluation process, and its features are explained in chapter 5. Details about the implementation of the IDE at the example of ExaSlang are given in chapter 6. It describes in depth how the implementation works and how it can be extended further to be adjusted to future development and other Domain Specific Languages.

Finally chapter 7 gives an overview of the work done, before in chapter 8 an outlook for further improvement of the implemented IDE as well as the possible realization of more complex features and other ExaSlang languages is given.

⁴<https://www12.informatik.uni-erlangen.de>

⁵<https://www.fau.eu/>

⁶<http://www.uni-passau.de/>

⁷<http://www.uni-wuppertal.de>

⁸<https://www.visualstudio.com/>

⁹<http://notepad-plus-plus.org/>

¹⁰<http://www.eclipse.org/>

¹¹<http://metaborg.org/spoofax/>

2 ExaStencil and ExaSlang

This chapter is in large part an excerpt from the articles *ExaStencils: Advanced Stencil-Code Engineering* [2] and *ExaSlang: A Domain-specific Language for Highly Scalable Multigrid Solvers* [3] from the year 2014 written by Christian Schmitt, et al. For further details on ExaStencils and ExaSlang please refer to these sources.

2.1 ExaStencils Overview

"The central goal of EXASTENCILS is to develop a radically new software technology for applications with exascale performance. [...] The aim is to enable a simple and convenient formulation of problem solutions in this domain. The software technology developed in EXASTENCILS shall facilitate the highly automatic generation of a large variety of efficient implementations via the judicious use of domain-specific knowledge in each of a sequence of optimization steps [...] The application domain chosen is that of stencil codes [...] Stencils codes are used for the solution of discrete partial differential equations and the resulting linear systems." [4]

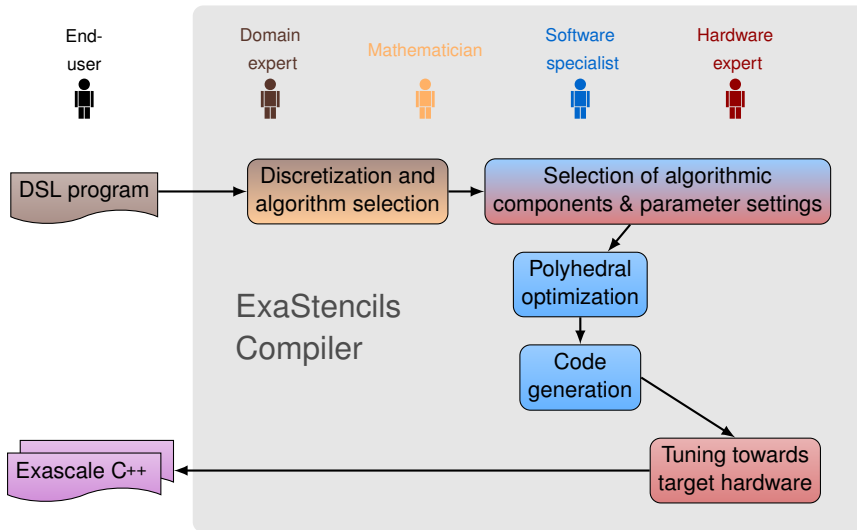


Figure 2.1: Concept of ExaStencils [2]

ExaStencils aims to provide all components shown in figure 2.1, from the DSL specifications, over compilers to knowledge libraries providing information of optimal algorithm

hardware combinations. This also includes the proper tools to access all of the offered functionalities, like the IDE implementation described in this thesis.

Currently ExaStencils is not available to the public, and therefore not yet under any license. This will most likely change when the package reaches its final completion. Although not set in stone, the current plan is to publish it under one of the common Free and Open-Source Software Licenses (FOSS Licenses) like the GNU Lesser Public License (GLPL).

2.2 ExaSlang Overview

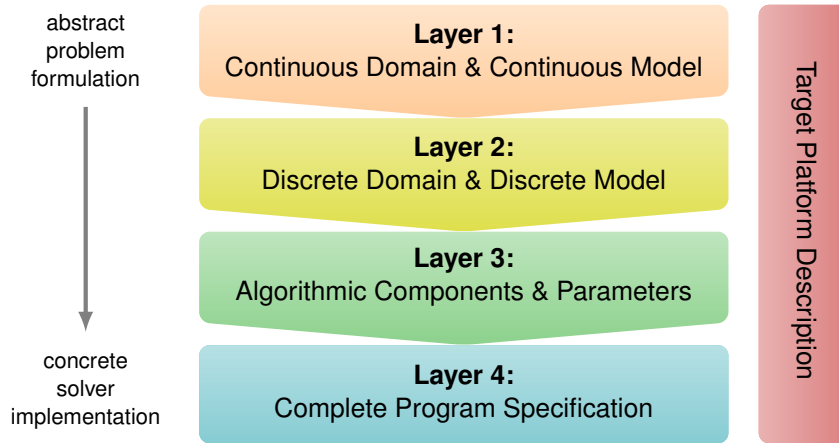


Figure 2.2: The DSL Layers of ExaSlang [3]

As part of ExaStencils the ExaSlang Languages provide DSLs for each of the optimization steps as shown in figure 2.2.

Layer 1 is aimed at physicists and engineers. It consists of constructs that are similar to physical formulas and expressions and describe the problem in a real world sense. The second Layer, aimed additionally at mathematicians, transforms the real world model specified in Layer 1 to a discrete, numerically solvable problem. While it is not an exact representation of the real world problem it still is mathematically exact at the discrete grid points and actually can be solved, while the continuous problem from Layer 1 usually can not.

Layer 3 allows to define algorithms that efficiently solve the discrete problem, appealing primarily to computer scientists and mathematicians. Finally, depending on the machine, problem and specifics about parallelization, the computer scientists can use Layer 4 to optimize the algorithm, especially by parallelization, from Layer 3 to be able to run at peak computation speed on the targeted machine. Layer 4 represents the complete program specification available to the user before the program, with all its components, is compiled and finalized for use.

The ExaStencils project not only provides the languages for each of these layers but also consists of transformation algorithms that translate the different Layers into each

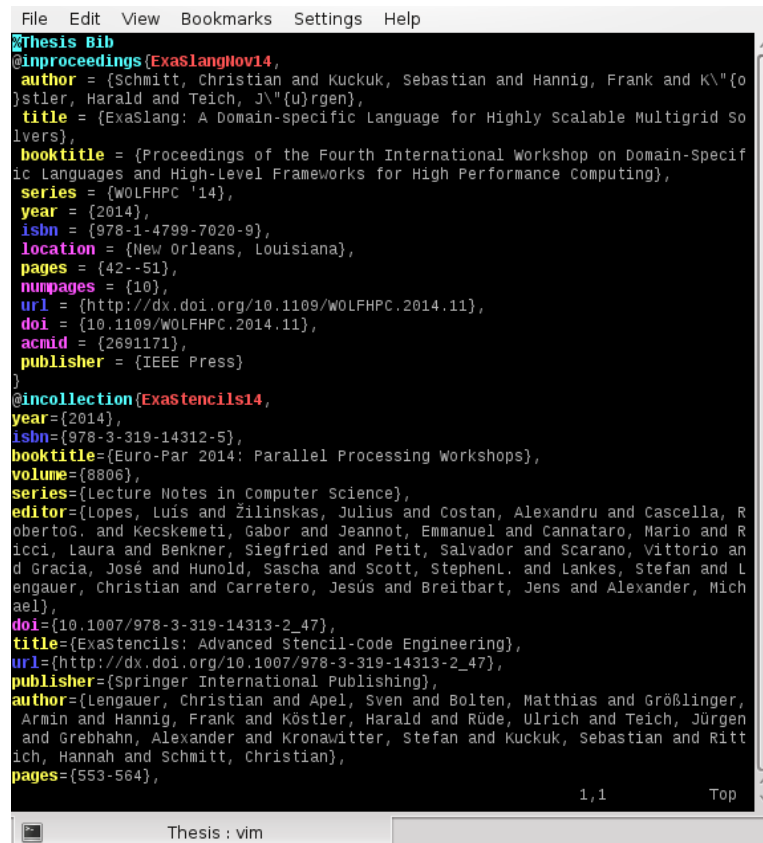
other without further input from the user. It also performs automated optimization steps between layers, depending on external information like the hardware specification file where possible. In a final step the Layer 4 file is translated to actual C++ code that can then be compiled and run on the target system.

Beside the four DSLs, one for each Layer, the hardware specification itself has a DSL. Additionally there are two configuration file types to guide the process. Overall there are, or will be, at least five DSLs in the ExaSlang language package from which one, the Layer 4 language, is already full-fledged. The Layer 3 language is currently under development.

All of the languages are external DSLs, meaning they are not based on existing programming languages but are completely self-sufficient. Internal DSLs in comparison are extensions of existing languages adjusted for a domain specific use. When the ExaStencils project was started, an evaluation of different tools for language design was conducted [5]. One result was that the advantage of having an already established IDE, which an internal DSL would provide, was not worth the effort to force the multigrid domain requirements into the corset of a general-purpose language. Therefore an external DSL approach was chosen. This tends to cause more work on every level of language development, especially the User Interface (UI) part, but it also offers the highest versatility concerning all of those aspects. This leads, in the end, to the best results possible.

3 IDEs and IDE Tools

3.1 Purpose and Capabilities of IDEs



```
File Edit View Bookmarks Settings Help
@Thesis Bib
@inproceedings{ExaSlangNov14,
  author = {Schmitt, Christian and Kuckuk, Sebastian and Hannig, Frank and Köstler, Harald and Teich, Jürgen},
  title = {ExaSlang: A Domain-specific Language for Highly Scalable Multigrid Solvers},
  booktitle = {Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing},
  series = {WOLFHPC '14},
  year = {2014},
  isbn = {978-1-4799-7020-9},
  location = {New Orleans, Louisiana},
  pages = {42--51},
  numpages = {10},
  url = {http://dx.doi.org/10.1109/WOLFHPC.2014.11},
  doi = {10.1109/WOLFHPC.2014.11},
  acmid = {2691171},
  publisher = {IEEE Press}
}
@incollection{ExaStencils14,
  year = {2014},
  isbn = {978-3-319-14312-5},
  booktitle = {Euro-Par 2014: Parallel Processing Workshops},
  volume = {8806},
  series = {Lecture Notes in Computer Science},
  editor = {Lopes, Luis and Žilinskas, Julius and Costan, Alexandru and Cascella, Roberto G. and Kecskemeti, Gabor and Jeannot, Emmanuel and Cannataro, Mario and Ricci, Laura and Benkner, Siegfried and Petit, Salvador and Scarano, Vittorio and Gracia, José and Hunold, Sascha and Scott, Stephen L. and Lankes, Stefan and Lengauer, Christian and Carretero, Jesus and Breitbart, Jens and Alexander, Michael},
  doi = {10.1007/978-3-319-14313-2_47},
  title = {ExaStencils: Advanced Stencil-Code Engineering},
  url = {http://dx.doi.org/10.1007/978-3-319-14313-2_47},
  publisher = {Springer International Publishing},
  author = {Lengauer, Christian and Apel, Sven and Bolten, Matthias and Großlinger, Armin and Hannig, Frank and Köstler, Harald and Rüde, Ulrich and Teich, Jürgen and Grebhahn, Alexander and Kronawitter, Stefan and Kuckuk, Sebastian and Rittich, Hannah and Schmitt, Christian},
  pages = {553-564},
  1,1 Top
```

Figure 3.1: Vim Editor¹ with an opened bibtex file in a classical console

Idea of IDEs

An Integrated Development Environment is supposed to provide the user of a programming language with a set of tools combined in a meaningful and conclusive way to aid him in developing programs. Such tools range from code editors, like the in figure 3.1 shown vim-editor, over GUIs for compilers to full-fledged programs that combine all

¹<http://www.vim.org/>

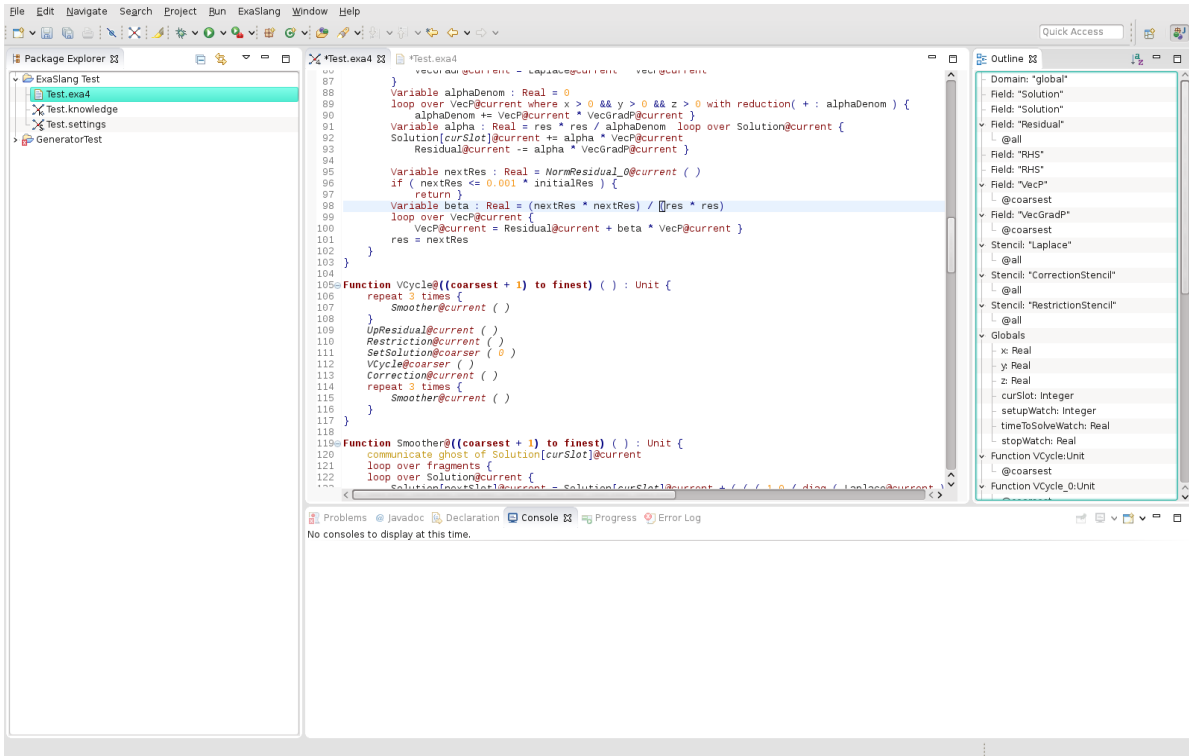


Figure 3.2: Example for an IDE: The Eclipse environment with an active ExaSlang Project

aspects of programming (concept to executable) in one cohesive and consistent environment. Prominent examples of such an environment are Microsoft Visual Studio and the Eclipse platform², the latter shown in figure 3.2. This thesis focuses mainly on the most basic but also most used IDE features, primarily a language specific text editor, accompanied by extension to the UI.

3.2 Editor Support

While for textual based languages any, even the most basic, text editor can suffice, a specialized editor for the programming language someone is writing in will speed up code writing, debugging and understanding considerably. To do so there may be any combination of the following main components of a language editor present.

3.2.1 Syntax Highlighting

The probably easiest and most notable form of editor support is syntax coloring. By giving different conceptual parts of the language different colors the structure and sub-structure can be presented on an extra level, compared to plain black on white text. As

²<http://goo.gl/s7iowq> - Google Trend Search - 2015/04/08

```

186 Function Solve ( ) : Unit {
187     UpResidual@finest ( )
188     Variable resStart_0 : Real = NormResidual_0@finest ( )
189     Variable res_0 : Real = resStart_0
190     Variable resOld_0 : Real = 0
191     print ( '"Starting residual at 0"', resStart_0 )
192     Variable totalTime : Real = 0

```

```

186 Function Solve ( ) : Unit {
187     UpResidual@finest ( )
188     Variable resStart_0 : Real = NormResidual_0@finest ( )
189     Variable res_0 : Real = resStart_0
190     Variable resOld_0 : Real = 0
191     print ( '"Starting residual at 0"', resStart_0 )
192     Variable totalTime : Real = 0

```

Figure 3.3: Screenshots comparing the same part of code once without and once with active Syntax Highlighting

is shown in the example in figure 3.3 the construct is divided into different parts that can be distinguished by the different colors and text styles quite easily. Compared to the non highlighted part of the example it is clearly visible that reading and understanding the code as well as navigating to different sections of the command is much easier with syntax highlighting. While with the use of an advanced text editor the user could do this manually, in case he wants to show his code to people unfamiliar with the language or code base, automation not only secures consistency, but also saves a lot of time better spent elsewhere.

3.2.2 Code Structuring and Bracket Matching

Another simple method of making reading and understanding of programs easier is the use of whitespaces, like space and tabulator, as mean to make indents of whole sections of the program. Often neglected or only halfheartedly done by the programmer, some programming languages, like Python³, enforce the use of indents for the program to work. Whether forced or voluntary, automatic indents and tools to correct misplaced indentations are a great way to make code more uniform and readable, especially in team environments, where different people with different personal preferences work on the same piece of code. Figure 3.4 shows the comparison of indented to non indented code in a rather extreme case. Additionally the highlighting of matching brackets in languages that use them to structure their code, like C++, is extremely helpful to quickly identify misplaced or missing brackets. An example of this functionality is shown in figure 3.5.

```

192 repeat until res_0 < 1.0e-8 {
193   numIt += 1
194   startTimer ( stopWatch )
195   VCycle@finest ( )
196   UpResidual@finest ( )
197   stopTimer ( stopWatch, totalTime )
198   resOld_0 = res_0
199   res_0 = NormResidual_0@finest ( ) }
200 stopTimer ( timeToSolveWatch, timeToSolve )
201 print ( '"Total time to solve in"', numIt, '"steps :"', timeToSolve )
202 print ( '"Mean time per vCycle: "', totalTime / numIt )

191 Variable numIt : Integer = 0
192 repeat until res_0 < 1.0e-8 {
193   numIt += 1
194   startTimer ( stopWatch )
195   VCycle@finest ( )
196   UpResidual@finest ( )
197   stopTimer ( stopWatch, totalTime )
198   resOld_0 = res_0
199   res_0 = NormResidual_0@finest ( ) }
200 stopTimer ( timeToSolveWatch, timeToSolve )
201 print ( '"Total time to solve in"', numIt, '"steps :"', timeToSolve )
202 print ( '"Mean time per vCycle: "', totalTime / numIt )

```

Figure 3.4: The same code, once with and once without proper indentation

```

176 Function InitRHS ( ) : Unit {
177   loop over RHS@finest {
178     RHS@finest = 0
179   }
180 }

```

Figure 3.5: Editor snippet showing Highlighting of matching brackets in Eclipse

3.2.3 Syntactical Error Detection

Syntactical error detection checks the typed code regarding consistency with the language definition and marks every position where the code can not be parsed into a valid representation of the language used. This ranges from the detection of simple typos, like the missing letters in figure 3.6, to recognizing if a section of code uses language constructs that are not valid at this position, but would be correct otherwise. Figure 3.7 shows the latter. When applied constantly during typing, syntactical error detection not only speeds up coding by preventing usually hard to recognize typing errors, but also helps learning the language by instantly remembering the user that he used a construct in the wrong way. This makes the feature exceptionally valuable for any DSL that is still in active development, as it will often change considerably with a new version [3], so that even a veteran user may have to learn parts of the language anew.

³<https://www.python.org/>

```

147 Function Restriction@((coarsest + 1) to finest) ( ) : Unit {
148     communicate gost of Residual@current
149     loop over fragments {
150     loop over RHS@coarser {
151         RHS@coarser = RestrictionStencil@curent * Residual@curent
152     }
153 }
154 }

```

Figure 3.6: Erroneously spelled keywords highlighted by Syntactical Error Detection

```

122 Function Smoother@([coarsest + 1] to finest) ( ) : Unit {
123     communicate ghost of Solution[nextSlot]@current
124
125     loop over fragments {
126     loop over Solution@current {
127         Solution[nextSlot]@current = Solution[curslot]@current

```

Multiple messages:

- Syntax error, not expected here: '['
- Syntax error, not expected here: '['

Press 'F2' for focus

Figure 3.7: Error detection found not expected literals

3.2.4 Syntactic Completion and Templates

```

71 exte
72
73 Func
74
75 }
76 Func
77
78
79
80
81
82
83
84
85
86
87
88

```


external Field Name < IDENT > ==> IDENT

external Field Name < IDENT > ==> Variable@Access

Figure 3.8: Template menu offering the choice of two different forms of the same construct after typing the first few letters

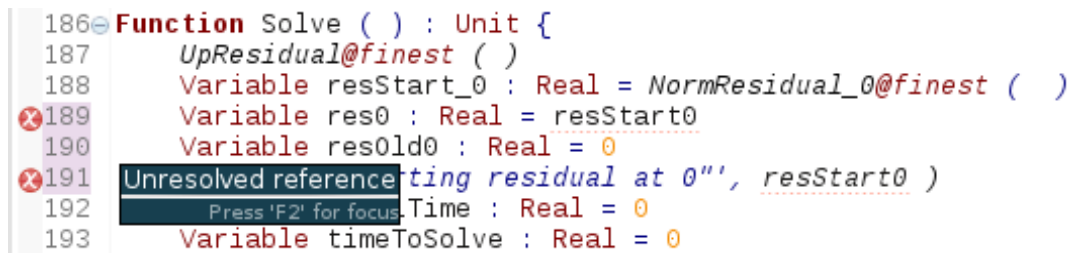
Similar to the previous feature, syntactic completion and templates help users to better navigate the language and avoid making annoying errors. Templates provide, based on the current editing position, a list of possible expressions allowed. In figure 3.8 such a list can be seen after the user has typed the beginning of an expression. While for often used expressions typing usually is faster then choosing from a potentially rather long list, the possibility to use them as quick references and to easily remember not so often used expressions make them an important feature of any IDE.

3.2.5 References, Name Resolving and Type Analyzing



```
75+ Function VCycle@coarsest ( ) : Unit {..  
78  
79+ Function VCycle@coarsest ( ) : Unit {..  
82  
83+ Function VCycle@finest ( ) : Unit {..  
86
```

Figure 3.9: Error when trying to redefine already existing function



```
186- Function Solve ( ) : Unit {  
187   UpResidual@finest ( )  
188   Variable resStart_0 : Real = NormResidual_0@finest ( )  
189   Variable res0 : Real = resStart0  
190   Variable resOld0 : Real = 0  
191   Unresolved reference'ting residual at 0'', resStart0 )  
192   Press 'F2' for focus:Time : Real = 0  
193   Variable timeToSolve : Real = 0
```

Figure 3.10: Editor reporting a non existent reference after a typo at the variable call

Most programming languages, including custom domain specific ones, have a form of individual object creation and naming by the programmer. Such names usually are custom strings, called Identifiers (IDENTs). Together with other parameters like object data types or function return types they form a footprint for each object. As those footprints usually have to be unique inside the scope or sub-scope of the program the editor can support the programmer by analyzing such definitions and displaying an error if a footprint appears more then once. On the other side, when an object is referenced in the code, the editor can check whether the referenced name is defined for the kind of object expected at the current position, and notifying the user if this is not the case. This is shown exemplarily in figure 3.10 More advanced analyses can not only check for IDENTs but also derive expected types and other parameters from the definition and apply those to the reference to check for consistency and correctness. They range from simply checking if return types of a function definition match the type of the variable returned in the function body, to completely analyzing expression chains for type consistency. While all of these capabilities are on the more complex side of editor features, the last one is certainly one of the most advanced tools in an IDE, especially when done at runtime during typing.

3.2.6 Dynamic Tooltips and Variable Templates

Beside for error checking the analyzing process, described in the previous paragraph, can also be used to offer additional information to the programmer. When the editor knows which of the custom objects are valid at any point of the program, it is only a short way to offer template like lists of, for example, variables to the user to choose

```

BasePath = new String();

DSL = new Folder();
CONFIG = new Folder();
GENERATED = new Folder();

```

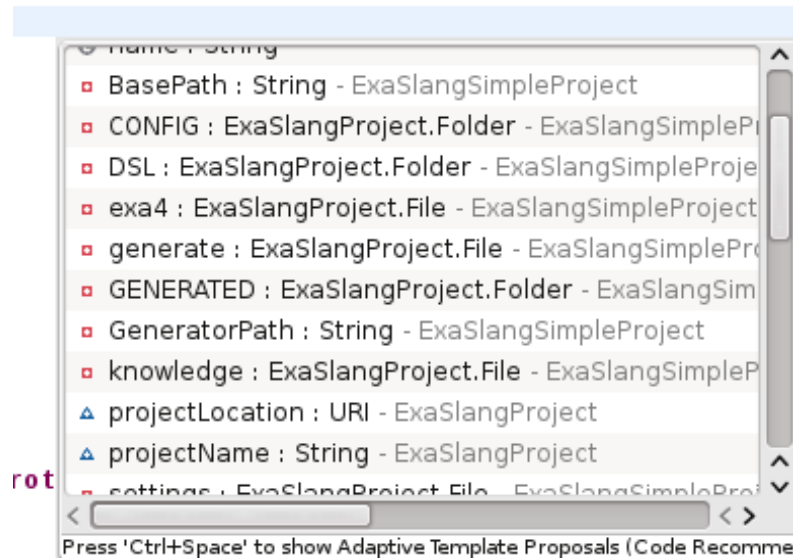


Figure 3.11: Choice of multiple variables in the template view

from. An example for such a list is shown in figure 3.11. Another option, although still difficult to automatically compose, is to offer context aware prewritten text info about the commands that can currently be used as seen in figure 3.12.

3.2.7 Outline Generation

Navigating code files with several thousand lines of complex and convoluted code is a real hassle. That is where the outline view comes into play. As shown in figure 3.13 on page 13 it usually consists of a list of top and near to top level definitions, that is considerably shorter and better readable then even collapsed code in the actual file. Another feature of outlines is the extraction of valuable information from single line definitions, that are loaded with syntactic literals. Examples are the return-type or the level definition in figure 3.13 from the ExaSlang Layer 4 Language. Besides saving search time for the single user, in multiple-user projects this feature is especially valuable because it offers a structured and static form of presenting code fragments to everybody. This makes different coding styles less of an error source than usually.

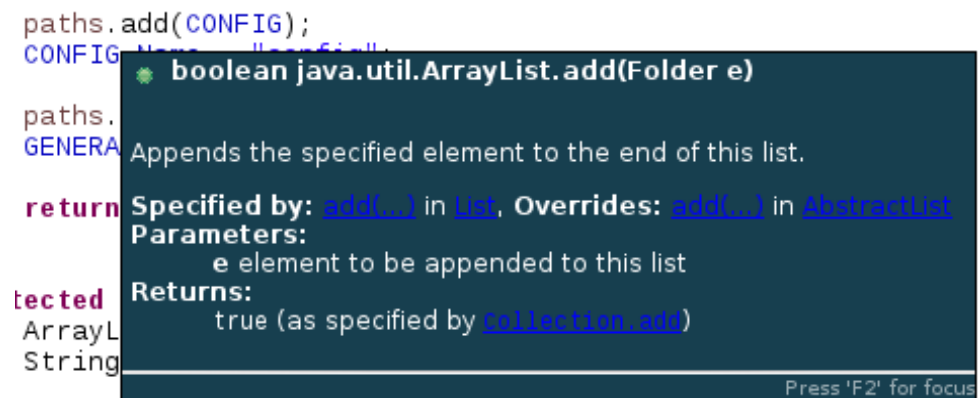


Figure 3.12: Dynamically generated tool tip based on the prewritten class information in the Java Library

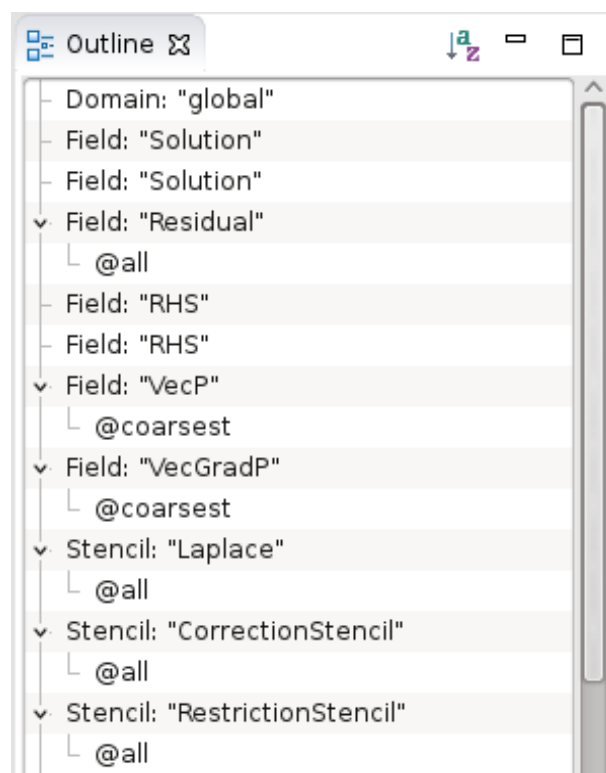


Figure 3.13: Outline view of an Exa4 file

3.3 Other IDE Features

Besides the features directly linked to the text editor, an IDE also can offer support to the developer in other sections of his project. While the features presented here certainly are among the most commonly found in all kinds of Integrated Development Environments, it is a fairly incomplete list as there is theoretically an endless number of such features, covering every thinkable and unthinkable use case. Features not covered here, are, for example, debugging and tracing support, in Eclipse and Visual Studio and integration of general helper languages, like Ant⁴ and Maven⁵ for Java used in Eclipse^{6,7}.

3.3.1 Project Templates

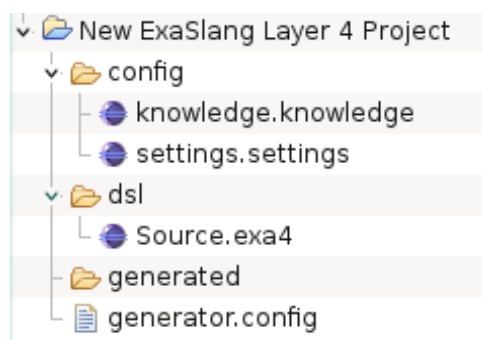


Figure 3.14: A newly generated ExaSlang project in Eclipse

For environments supporting many different languages and project types it often is helpful to offer the user templates for specific projects to start from. This allows the language creator to offer basic setups and integration of other elements of the environmental framework to be setup properly without the user having to learn these, often only briefly needed, parts. Additionally, by providing a basic folder and file structure at the beginning of a new project examples and manuals can refer to those templates to better explain basic relations of the different parts of the project. Such a generated structure can be seen in figure 3.14. And, of course, even the experienced user profits from not having to do the same basic steps every time he starts a new project.

3.3.2 Code Compilation and Generation Support

Usually programs that are under development need to be tested and debugged. This can either be done in an external program, the system console, or be integrated into the IDE. The latter offers the advantage of not having to switch between different windows and applications as well as the possibility to show errors and debug information directly

⁴<http://ant.apache.org/>

⁵<http://maven.apache.org/>

⁶<http://www.eclipse.org/eclipse/ant/>

⁷<http://www.eclipse.org/m2e/>

inside the familiar coding environment. When combined with a good UI to configure the actual compilation process and integrating external programs, for example by redirecting their output to an IDE internal console, the advantages are even greater.

3.3.3 Interface and Workbench Adjustments

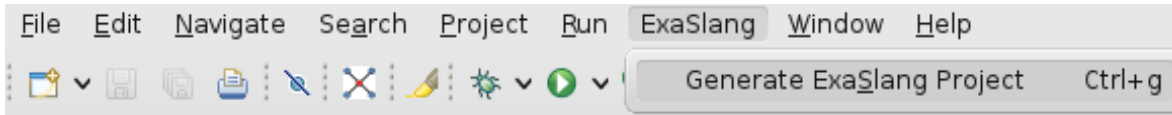


Figure 3.15: New Toolbar Icon and Options Menu for ExaSlang in Eclipse

Usability of an Integrated Development Environment is strongly linked to the quality of the User Interface it offers to the end user [6]. While in a completely custom created environment this can be achieved by designing the UI exactly to the needs of the supported language, when using a general IDE framework like Eclipse it is, therefore, important that the UI can be and is adjusted to the individual needs of the project and language implemented. Those adjustments can for example be the existence of outline views or the exact look of the project structure. They also include additional menu and shortcut buttons to language specific functions, as shown in figure 3.15 that otherwise would need the use of a console to be used.

4 Evaluation of LWBs and similar Tools regarding IDE support

Language Workbenches (LWBs) are an approach to offer tools and frameworks to ease the development of programming languages in general and domain specific languages in particular.

Description of the Evaluation Process

To evaluate which of the Tools and LWBs shown in table 4.1 is best suited to provide an IDE for an already defined external DSL, a list of several requirements has been composed. These differ in importance, from must-have to nice-to-have. Section 4.1 describes these requirements in detail.

After the list had been established, the actual evaluation process began. For details of the results of this process, refer to section 4.2. A first iteration revealed which of the candidates did not fulfill all of the must-have requirements. Those were removed from the pool. The remaining candidates were then examined closer with regard to soft requirements until only a few languages remained.

In the last step, ExaSlang Layer 4 was partially implemented in those tools to test their feasibility for the intended use.

In the end, the best suited candidate for the formulated requirements was chosen as the winner.

Tools in the Evaluation

Notepad++	Monticore	Spoofax
Eclipse	MPS 3.1	MetaEdit+
XText	SugarJ	Rascal
Onion		

Table 4.1: List of all LWBs and Tools examined

4.1 Requirements

The requirements have been divided into three categories. Those categories are general, editor and others.

Requirements themselves are rated between must-have, or hard requirements, over soft requirements to nice-to-have ones. Table 4.2 shows all of them, divided by category and plotted on a scale according to their importance. Hard requirements for example are licensing restrictions, while soft requirements usually are the support for specific editor features that should be supported, like syntax coloring. Nice-to-have requirements are things like outline generation and an easy installation process for the end user.

Group	Importance		
	hard-requirement	soft-requirement	nice-to-have
General	Platform Independent		
		Easy Deployment	
	Free of Charge Open License		
Features		Syntax Highlighting	
		Syntactical Error Detection	
		Reference and Name Resolving	
		Completion Templates	
		Auto Indents	
		Folding	
		Variable Templates	
		Dynamic Tooltips	
Others			Outliner
		Documentation	
		Stability of Environment	
		Automated Generation	

Table 4.2: List of all Requirements sorted by importance

4.1.1 General Requirements

The ExaSlang project itself is deemed to be platform independent. Therefore the ExaSlang IDE must be platform independent as well, forming the first hard requirement. Secondly, to support the approach of ExaSlang becoming a widely used standard language for multigrid solver programming, its IDE should be easy to deploy to the end user. This means, no, or almost no, effort besides visiting the ExaSlang homepage and downloading the IDE, or something similar, should be necessary to use it. Being somewhat vague in its formulation, this is a semi hard requirement. Finally, as mentioned at the end of section 2.1 the legal base of ExaStencils, and there-

fore ExaSlang, is not yet determined. This means, any possible restrictions regarding the legal status of collaborating software should be avoided. Therefore the license of all used tools and libraries involved with the ExaSlang IDE must allow closed source contribution and deployment of components while still being free of charge. Of course, this makes the licensing requirement a must-have one, as well.

4.1.2 Editor Requirements

While in theory, the goal for the editor is to have every, or at least as many of the capabilities introduced in section 3.2 as possible, there certainly are features that are must-haves and others that are nice-to-have, but not mandatory. Nonetheless, all editor features are considered soft requirements, as they will not prevent the IDE from doing its job, as failure of the hard requirements, mentioned in the previous section, would do. Syntax highlighting, as well as syntactical error detection, are considered quasi must-haves. They provide the greatest improvement over a plain text editor and are universally helpful for all users at all times. Nobody is safe from making typing errors once in a while, and having different colors appear in the written text helps greatly to quickly see if written code will be recognized by the compiler later on.

A similar effect, but not as important, primarily because it is a more complex feature, is reference and name analyzing and resolving. The typing error argument stands here as well, accompanied by relieving the user from having to memorize the correct spelling of dozens or more variable and function names. Especially upper- and lowercase errors often result in confusion and can easily be prevented by this feature.

On the other side, completion templates, automatic indents as well as folding capability, are still helpful in many circumstances, but the lack of them is hardly a hindrance for effective coding. Still, they are of great help in situations where the user has to read and understand foreign code, as they, like paragraphs and chapters in regular texts, help to recognize structure.

Dynamic tool tips and variable templates are, from a programmer's point of view, really great productivity increasing features. They help preventing erroneous reference even better than name resolving and are, obviously, also a great relief for the memory of the programmer. From an IDE's point of view this is perhaps the most difficult feature to properly implement, as it requires the editor to understand the language on runtime far beyond the pure syntactic level. Overall, this brings those features to a nice-to-have position in the requirements list.

Least important, as usually only regarded as a convenience feature, is outline generation. Serving as an addition to functions other features already offer without introducing unique functions itself, it is purely nice-to-have.

4.1.3 Other Requirements

Besides the features the IDE has to offer on the user side, there also are some requirements on the creator side.

Foremost, as it is planned to support the ExaSlang IDE over at least several years, it

is important that the resulting implementation can be read and understood by many different people making contributions easily. Especially direct advice from the previous contributors should not be required, as this is not feasible in the given academic environment. In detail, this means documentation and support of the framework LWB should be existing, helpful and complete. The environment should be stable enough to depend on the current implementation working even in future iterations without too much interference, as well. While those points are rather vague and often can not be clearly specified for every tool, at least a rough estimation should be used as point of comparison.

Secondly, as the ExaSlang language is under constant development and changes almost weekly the support effort to keep the IDE up to date should be minimal. This is realized, either by offering templates and simple instructions for expanding and changing it or, ideally, by offering programs and scripts that implement most changes on their own, without overhead supervision by the user. The ability to fulfill this requirement was, due to the lack of time to fully implement all features in the tested workbenches, only estimated. Nonetheless it has not been neglected for the evaluation.

4.2 Evaluation Process

In section 4.2.1 the first two evaluation steps are summarized. For step one only the reason for the elimination is given. Those eliminated during the second step are shortly described with the reason for their failure. The tools that made it to the final step of the evaluation process are described in length and then compared in section 4.2.2, before the overall results are presented in section 4.2.3.

4.2.1 First and Second Evaluation Steps

Examined tools that did not fulfill the basic hard requirements were, Notepad++, Onion, MPS 3.1 and MetaEdit+. The first two are not platform independent as they are only available for Windows. MPS 3.1, is not open source and MetaEdit+ does not provide textual editor support. Table 4.3 shows an overview of the examined tools and the evaluation step they reached.

Eclipse¹ itself offers many tools that allow the implementation of own editors and therefore support for DSLs, but the evaluation process has shown that those features often are already used by LWBs as a basic framework. Therefore it was concluded that those should be used instead of a plain Eclipse implementation.

SugarJ² was looked at after appearing in the LWB challenge 2013 [7]. It did not fit the requirements, because it is more of a meta programming tool to combine existing

¹<http://www.eclipse.org/>

²<https://www.student.informatik.tu-darmstadt.de/~xx00seba/projects/sugarj/>

languages than a tool to implement new languages . Also, SugarJ could not be made to run reliably on the development working station.

Rascal³ simply did not offer any automated generation of editor features and was therefore deemed to be not fit enough for further evaluation.

Tools in the Evaluation

Notpad++	Monticore	Spoofax
Eclipse	MPS 3.1	MetaEdit+
XText	SugarJ	Rascal
Onion		

Table 4.3: Tools and LWBs, background shaded after evaluation step they reached. Darker means earlier elimination.

4.2.2 Final Evaluation Step

Monticore⁴

Developed by the RWTH Aachen University, Monticore is unique in the way it handles project generation. It does not generate locally on the users machine but provides a server that does all of the generation work, a so called Online Software Transformation Platform (OSTP). Its focus lies on efficient and fast implementation of DSLs and their tools.

The creators of Monticore offer a fairly complete documentation of their concepts as well as a wide variation of examples for the version 2.2.0 of their product, dating back to the end of 2013 [8]. For newer versions of Monticore there is no documentation at all. The latest known released version from late 2014 is 3.2.0. As the download servers for Monticore can currently, as of 2015/04/07, not be reached, it is unknown if a newer version has been released.

The definition language Monticore uses is closely aligned to the Extended-Backus-Naur-Form (EBNF) [9] with components of object oriented languages similar to Java.

```

1 Field = "Field" Name "<" Name "," Name "," FieldBoundary ">" ("["
   IntegerLit "]" )? (Level)?;
2
3 FieldBoundary = (BinaryExpression | "None");
4
5 interface IndexGroup;
```

³<http://www.rascal-mpl.org/>

⁴<http://www.monticore.org/>

Feature	Monticore	XText	Spoofax
Platform Independence	yes	yes	yes
Ease of Deployment	easy	medium	easy
Free of Charge	yes	yes	yes
Open License	yes	yes	yes
Ease of Implementation	very easy	medium	easy
Syntax Highlighting	yes	yes	yes
Syntactical Error Detection	yes(?)	yes	yes
Reference and Name Resolving	yes(?)	yes	yes
Completion Templates	yes	yes	yes
Automatic Indents	yes(?)	yes(?)	yes
Folding	yes	yes	yes
Variable Templates	no(?)	yes	yes
Dynamic Tooltips	no(?)	yes	limited
Outline Generation	yes	yes	yes
Documentation Quality	good	bad	medium
Documentation Quantity	very good	medium	good
Stability during development	medium	good	medium
Stability when deployed	unknown	good	medium
Integrated Feature generation	medium	very bad	medium
Possibility of automated IDE generation	good	bad	medium

Table 4.4: Feature overview for candidates of the final evaluation step, features marked with „?“ could not be tested but documentation lets assume their existence

```

6 interface Index extends ("[" IntegerLit)=> IndexGroup;
7 DoubleIndex implements Index = "[" IntegerLit "," IntegerLit ";
8 TrippleIndex extends ("[" IntegerLit "," IntegerLit ",")=> DoubleIndex
  = "[" IntegerLit "," IntegerLit "," IntegerLit ";

```

Snippet 4.1: Monticore language definition

Supported editor functions by Monticore are outline, folding and keyword highlighting, as well as customizable tool tips and content completion. Those features are mostly defined directly in the language definition file and then automatically generated [8].

```

1 concept editorattributes {
2   keywords:
3     Func, Function,
4     Var, Variable,
5     Val, Value,
6     Domain, Layout, Field, Stencil, StencilField, Set, external,
      Globals,

```

```

7   repeat, times, count, with, contraction,
8   loop, until, over, fragments, where, starting, ending, stepping,
    reduction,
9   if, else,
10  current, coarser, finer, coarsest, finest, to, not, all, and,
11  with, communication, None,
12      apply, bc, to,
13      begin, finish, communicate, communication, dup, ghost, of,
14      diag,
15
16  foldable:
17      Programm,
18      Definition,
19      LevelsListe,
20      Function,
21      FuncCall,
22      StatementClause,
23      CommunicateStatement,
24      LayoutOptions,
25      Stencil;
26 }

```

Snippet 4.2: Editor function definition in Monticore

XText⁵

XText is an open source project, led and managed by itemis⁶, and part of the Eclipse.org Project. This section is based on information from the XText homepage⁵ and experiences made during the evaluation implementation of ExaSlang.

Called a „language development framework“ by the developers, it provides support to generate user experience similar to Java programming. Naturally, as part of the Eclipse.org Project, it is completely realized inside of the Eclipse environment. At the date of writing, the current version is 2.8.0 released on 2015/03/11. The version evaluated for this thesis is 2.7.3 from 2014/11/20.

The main features XText provides are syntax coloring, content completion, strong type validation and runtime error detection, as well as a superior Java integration and the interconnection to other Eclipse tools.

The language definition in XText is extremely close to what an EBNF would look like, and with the direct and required integration of variables in the definition makes it easy to structure the language.

```

1 loopOverFragments:

```

⁵<http://www.eclipse.org/Xtext/>

⁶<http://www.itemis.com/>


```

2  'loop' 'over' 'fragments' ('with' clause = reductionClause)? '{'
    (list += statement)+ '}'
3  ;
4
5  loopOver returns loop:
6  'loop' 'over' field = fieldLikeAccess
7  ('sequentially')?//FIXME: Likely to change in the future (HACK)
8  ('where' expression = booleanexpression)?
9  ('starting' exprIndex += expressionIndex)?
10 ('ending' exprIndex += expressionIndex)?
11 ('stepping' exprIndex += expressionIndex)?
12 ('with' reduce += reductionClause)?
13 '{' list += statement+ '}'
14 ;
15
16 reductionClause:
17 'reduction' '(' left = (IDENT | PLUS | MULTIPLY ) ':' right = IDENT
    ')',
18 ;

```

Snippet 4.3: Example of XText's language definition syntax

All additional functionalities are directly coded in Java, and therefore should be easily understandable for new users. Unfortunately this is not the case. XText is a prime example why DSLs are necessary. Its strong dependency on Java and Java like code add up to hundreds of files classes and functions being generated, as can be seen in figure 4.1. These provide the basic framework the user has to work upon. This makes the whole process rather confusing and convoluted, and definitely generates a lot of overhead code that could be avoided.

Although the project is under longtime development, the documentation, while fairly complete feature wise, is confusing and often incomplete when it comes to details, like where to put or find specific files or how naming conventions work in the generated files.

```

1  /**
2   * Returns the value of the '<em><b>Size</b></em>' attribute.
3   * <!-- begin-user-doc -->
4   * <p>
5   * If the meaning of the '<em>Size</em>' attribute isn't clear,
6   * there really should be more of a description here...
7   * </p>
8   * <!-- end-user-doc -->
9   * @return the value of the '<em>Size</em>' attribute.
10  * @see #setSize(String)
11  * @see exaSlang.exaSlang_L4.ExaSlang_L4Package#getdatatype_Size()
12  * @model

```

```

13  * @generated
14  */
15  String getSize();

```

Snippet 4.4: Example of badly done comments and documentation in XText. Lines 4 to 8 show this in particular.

This makes it hard to start working on specific features without completely learning the language theory wise beforehand. Emphasized by the lack of individually automatically generated functions and features this is a real time drain. Although, this might somewhat be mitigated by the fact, that the developers of XText offer additional courses and support to finance the XText project.

As mentioned before, the integration into the Eclipse framework really is exceptional, but it still leaves room for improvement. For example, while XText prepares a feature project for the language, it does not offer an update site to accompany the deployment automatically.

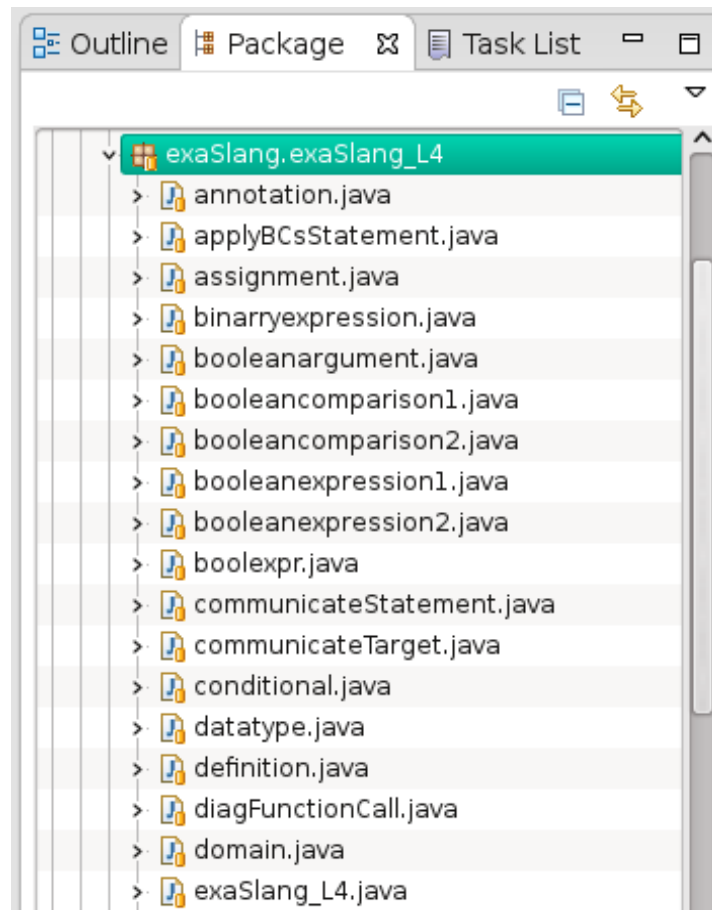


Figure 4.1: Excerpt of generated classes for the test implementation of Exa4

Spoofax⁷

Spoofax is a long standing project from the University of Delft, dating back to at least the year 2007 [10]. It is still under heavy development, going from version 1.2 in August 2014 to 1.4 in March 2015 [11]. The version evaluated was Spoofax 1.1, as 1.2 was fairly new and had problems with lifting the Syntax Definition Formalism (SDF) syntax from SDF2 to SDF3 for the evaluation implementation. As with the release of version 1.3, in November 2014, this problem, discouraging the use of version 1.2, has been fixed, a switch to the newer version has been done during the evaluation process. Because of this, the examples in this section usually consist of two parts showing both version 1.1 and 1.3. More detailed information about Spoofax can be found at metaborg.org/ in general and metaborg.org/spoofax/ in particular.

Core concept of Spoofax is the usage of different languages to define different parts of the DSL and the corresponding editor, that are implemented.

The language definition itself is written in the Syntax Definition Formalism. While still close to the EBNF, it deviates significantly from this standard formalism in some points, especially in the syntax it uses.

```
1 StatementInsideRepeat* ReturnStatement -> RepeatBody
   {cons("RepReturn"), prefer}
2 StatementInsideRepeat* -> RepeatBody {cons("RepNoRet")}
3 Statement -> StatementInsideRepeat
4 BreakStatement -> StatementInsideRepeat
5 "loop" "over" "fragments" "{" SectionBody "}"
6 -> LoopOverFragments {cons("LoopOverFragments")}
7 "loop" "over" "fragments" "with" ReductionClause "{" SectionBody "}"
8 -> LoopOverFragments {cons("LoopOverReductionFragments")}
```

Snippet 4.5: Spoofax language definition in SDF2

With the introduction of the third iteration of the Syntax Definition Formalism, SDF3, in Spoofax version 1.2, this difference has been reduced to some extend.

```
1 RepeatUntil.RepeatUntil = "repeat" "until" SimpleComparison "{"
   RepeatBody "}"
2   RepeatBody.RepReturn = StatementInsideRepeat* ReturnStatement
   {prefer}
3   RepeatBody.RepNoRet = StatementInsideRepeat*
4   StatementInsideRepeat = Statement
5   StatementInsideRepeat = BreakStatement
6
7 LoopOverFragments.LoopOverFragments =
8   "loop" "over" "fragments" "{" SectionBody "}"
9 LoopOverFragments.LoopOverFragments_reduction =
10  "loop" "over" "fragments" "with" ReductionClause "{" SectionBody "}"
```

⁷<http://metaborg.org/spoofax/>

Snippet 4.6: Spoofax language definition in SDF3

The documentation for SDF is widely complete, but there is no update documentation for SDF3. The only help offered for the new version is a short description of concepts. And the general tip to keep using the old documentation.

To implement the editor functions, it uses a variety of languages that are all derived from the Stratego transformation language, which builds the core of Spoofax itself. Like SDF3, those languages lack a proper documentation as only explained examples are given on the websites of each language, as can be seen in figure 4.2. Nonetheless for most editor functions examples and automatically generated functionality is provided, which helps greatly to resolve those shortcomings. Spoofax offers a complete set of edi-

Definition Sites

The following rules declare definition sites for module and entity names:

```
rules
  Module(m, _): defines non-unique Module m
  Entity(e, _): defines unique Entity e
```

The patterns in these rules match module and entity declarations, binding variables `m` and `e` to module and entity names, respectively. These variables are then used in the clauses on the right-hand sides. In the first rule, the clause specifies any term matched by `Module(m, _)` to define a name `m` in the `Module` namespace. Similarly, the second rule specifies any term matched by `Entity(e, _)` to define a name `e` in the `Entity` namespace.

Figure 4.2: Excerpt from the Name Binding Language documentation website⁸ showing an explained example of the Definition Sites functionality

tor features, ranging from coloring and folding over outline to type analysis and content completion.

Also, while not relevant for this project, but still notable, it offers support for transforming any code written in the newly defined DSL into other languages, like Java or C++, when corresponding definitions are implemented.

The support for integration in Eclipse is minimal consisting of a pregenerated plugin.xml file and a simple integration of toolbar menus into the User Interface, which is shown in figure 4.3 in its state when Spoofax uses it for its internal languages.

⁸<http://metaborg.org/nabl/>

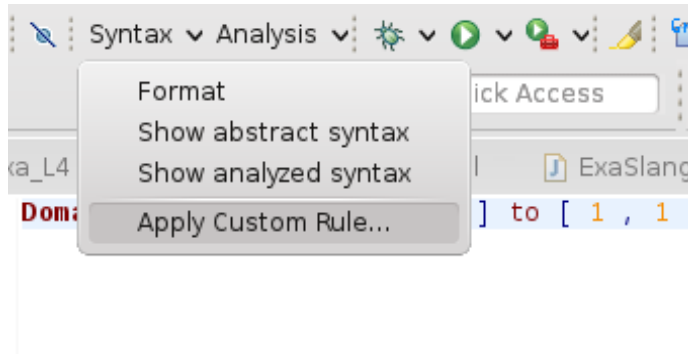


Figure 4.3: Spoofax toolbar menu used by the Spoofax environment itself

Comparison of the three Candidates

Table 4.4 shows an overview of the features present in the free final candidates as well as, where appropriate, a rating between „very good“ and „very bad“ on a five step scale. While fulfilling most of the requirements pretty well, the ExaSlang implementation in Monticore could, despite a week long effort, not be made working at all. Disqualifying an otherwise good, maybe even winning, candidate. The errors appeared one at a time, seemingly overshadowing each other in a way that makes it almost impossible to comprehend what is happening. For example, removing a part of the language, that was reported as faulty let a new error in a completely different section of the language appear. The new error was neither mentioned before nor was it connected logically to the removed part.

This leaves only XText and Spoofax as possible candidates for the case study implementation. While XText takes the lead in GUI integration and stability, Spoofax is not that far behind. While Spoofax and XText are on par editor feature wise, Spoofax clearly has the advantage due to the ease of implementation. The better documentation and the automated generation of features clearly see Spoofax as favorite, with XText being an almost total failure in the latter category.

4.2.3 Results

Overview

Overall, eleven different LWBs and IDE tools have been evaluated. In general, the results were rather mixed and not totally satisfying. While the number of tools and approaches is plentiful, most of them are either underdeveloped or tailored to such specific use cases that usability was rather limited. Only the three workbenches examined in detail during the last evaluation step provided a general purpose support for the intended usage. No workbench fulfilled all of the soft requirements to a totally acceptable degree. Nonetheless, a workbench, while not perfect still suitable for the provided problem has been found.

Conclusion: Spoofox and Eclipse

The only possible conclusion of the evaluation, given the presented results, is the use of Spoofox inside the Eclipse environment.

It not only fulfills all of the hard requirements with ease, but also has the best score on the soft requirements overall. Especially the amount of offered editor features, and the, although still under heavy development, automated generation of said features are remarkable. Combined with the, compared to most of the other candidates, rather good, although somewhat outdated and short, documentation the overall results were satisfying.

Furthermore, the evaluation has shown that most LWBs do not offer a complete environment themselves but are integrated in other environments, most notably Eclipse. This led to the additional conclusion that this approach should be taken by the ExaSlang implementation as well. As Eclipse is widespread, multi platform and free of charge, it fits perfectly in with the hard requirements and provides potential users with a generally well known and often familiar User Interface structure.

4.3 Remarks on the Evaluation Process

With the reports [7] released from the yearly **Language Workbench Challenge**⁹ held as a workshop during the **Code Generation Conference**¹⁰ in Cambridge and the paper **An evaluation of domain-specific language technologies for code generation** [5] there was a pretty solid base to work from. This is especially true for the initial choice of LWBs to examine. Nonetheless, the differences in requirements between the approach in these papers and the approach shown in this thesis made another evaluation inevitable. Both sources used not existing, or only in theory existing Domain Specific Languages as starting points, while this thesis examines whether it is possible to transfer existing DSLs into LWB environments.

This kind of approach seems not to be covered by existing papers extensively and therefore it was difficult to determine which requirements to set. In the end, the chosen requirements led to promising results, indicating that they can be reused for future evaluations of similar kind. The biggest issue with the presented evaluation process was the time constraints, about 160 working hours were invested overall. Although, given the fact, that the mentioned evaluations of LWBs were usually conducted by several people [5] or even a whole team [7] for each workbench, this still was surprising. This amplifies the assumption that a multi person approach is preferable to the used single person approach. This of course would have to take into account the different skill levels of multiple persons. Therefore objective parameters have to be used or a weighing of the cons and pros has to be done.

The time constraints limited the overall number of candidates, especially in the second and third phase, after the tools not fitting the hard requirements were eliminated. Meaning that, despite the process resulting in a successful search, other, better, results might

⁹<http://www.languageworkbenches.net>

¹⁰<http://codegeneration.net/>

have been overlooked, simply because they were not obvious during the second process stage. Still, Spoofax proved to be a solid and good choice and can be recommended for IDE implementation of existing DSLs.

5 The Spoofax Language Workbench

The Spoofax Language Workbench¹ was and is being developed at the Delft University of Technology. Its current release version is 1.4 from 2015/03/06. The version used and described in this thesis is 1.3.1 from 2014/12/09. It is part of the MetaBorg² meta-programming tools collection and serves as platform between the different languages and tools of MetaBorg to enable them to work together as a Language Workbench for textual DSL development. The LWB is designed for Eclipse 4.3 to 4.4 with Java 7 and supports the creation of fully featured Eclipse editor plugins [11]. Installation instructions can be found in the „Getting Started“ section of the Spoofax Tour page³.

Different Languages for Different Tasks

As part of a meta language and tool environment the Spoofax Language Workbench consists of several Domain Specific Languages for different parts of the language and editor creation. These languages are Syntax Definition Formalism v.3, Name Binding Language, Type Specification Language and Stratego. Additionally it uses the Editor Services Language for its editor integration and the Spoofax Testing Language for language-agnostic tests on the implemented DSL [11].

This chapter will describe these languages in general and give detailed examples if this is not done in section 6.1. The descriptions are based on their respective counterparts on the metaborg.org website [11] and the experiences made during the implementation of ExaSlang.

General Structure

All languages in Spoofax are divided into modules, that are subdivided into sections. Modules are usually used to better distinguish between different parts of a language or to reuse parts of it in other languages. Every module definition in Spoofax has to be defined in its own file. To use one module in another the `include` section is used. Listing 5.1 shows how the `ExaSlang_Base` module is included at the beginning of the `Exa4` module.

```
1 module Exa4
2
3 imports
```

¹<http://metaborg.org/spoofax/>

²<http://metaborg.org/>

³<http://metaborg.org/spoofax/tour/>

Snippet 5.1: Example of a module being included in another module

5.1 The Syntax Definition Formalism v.3 (SDF3)

The Syntax Definition Formalism has been under development since at least 1989 when a reference manual for the SDF was released [12]. While SDF3 has some notable differences syntax wise to the original SDF, its concepts are still quite similar and therefore, in lack of a full documentation for SDF3 it is advised, by the creators, to use the SDF2 documentation⁴ instead. SDF3 has been introduced with Spoofax 1.2 in August 2014 and an explanation of its concepts can be found at metaborg.org/sdf3.

The primary syntactical change was the order productions are written in, the integration of constructors more directly into the language, as can be seen by comparing listing 4.5 to listing 4.6 on page 25. Additionally template definitions have been newly introduced.

Lexical and Context-Free Syntax

SDF3 differentiates between lexical and context-free syntax. Both define productions and are handled in almost the same way by most of the other components of Spoofax. The only two differences are that lexical productions can not have constructors and can not include context-free productions, while context-free productions can have constructors and include both types of productions. Usually lexicals are used to define low level elements of a language like Keywords and Layout symbols. Context-free productions are used to define all other parts of the DSL.

Productions: Sorts and Constructors

Every definition or expression of a language component in SDF3 is called a production. These productions are identified by two different parts. The first part is called sort, the second constructor. They are written with a dot between them as presented in the example below

```
Sort.Constructor = Definition-content {Flags}
```

After the declaration a equal sign follows which divides the declaration part from the definition part of the production. Here the actual language definition is done much like it would be in the EBNF. Additionally the production can have certain flags that are written inside of curly brackets. When multiple flags are used they are separated by a comma.

Sorts are part of all productions. When the production consists of any other parts than pure literals, or the sort has more than one definition, it should be predeclared in the sorts section of the grammar. Otherwise, some editor functions do not work properly. Sorts can have multiple definitions bound to their name, as shown in the example below

⁴<http://homepages.cwi.nl/~daybuild/daily-books/syntax/2-sdf/sdf.html>

where the sort `Mine` can be the literals "mine", "my" and "not yours". Sorts always begin with a capital letter.

```
1 module myLanguage
2     sorts
3         Mine
4         Language
5         Something
6     lexical syntax
7         Mine = "mine"
8         Mine = "my"
9         Mine = "not yours"
10        Something = "everything"
11    context-free syntax
12        Language.combination = {Mine "and"}+ {prefer}
13        Language.single = Mine {avoid}
14        Language = "yours"
```

Snippet 5.2: Exemplary language definition

Constructors are not required and are only necessary if a sort consists of non static elements, usually other sorts. As can be seen in the example above a single sort can have multiple constructors assigned to it. For better readability it is advised to define a constructor for every production. Technically constructor names do not have to be unique, but as they are referenced in the editor definitions quite frequently it is strongly advisable to only use a constructor name once.

List of Common Flags

- `{prefer}` production is preferred to other productions if they are disambiguous
- `{avoid}` production is avoided if disambiguous with other productions
- `{reject}` production is never valid if disambiguous (usually used in lexicals to distinguish between IDENT and keywords)

A list of all possible flags can be found at metaborg.org/sdf3/ in the `Attributes` section.

Definition of Lists

Lists are defined by surrounding the part of a production that acts as a list with curly brackets, followed by either "+" or "*" denoting if there has to be at least one entry ("+" in the list or if it can be empty ("*")). Inside the brackets the first element defines the items in the list and the second element defines the means by which items are divided from each other. In the example above `Language.combination` is a list of

Mines separated by the literal **and**. Note that the separator can only appear between two items, so a list always ends with an item.

Regular Expressions of Literals

In lexical productions it is possible to define sequences of literals that are allowed in the production. This is particularly useful when defining parts of the language that consist of infinite amounts of possible combinations of symbols. Usually this is the case for IDENTs and Numerals. The IDENT definition in listing 5.4 shows how a regular expression of characters looks like in SDF. In this case an IDENT can consist of any combination of letters, numbers and the `_` symbol.

Start Symbols

Start symbols can be any production either lexical or context-free and serve as starting points for the Abstract Syntax Tree (AST) parser. So, only productions reachable from a start symbol are part of the grammar that is usable in a program later on. There can always be only one AST per file. That means productions from separate trees may not occur in the same file without causing parsing errors.

```
1 lexical start-symbols
2     Something
3 context-free start-symbols
4     Language
```

Snippet 5.3: Start-symbols for example language

In this example a file can either contain any of the „Language“ productions or the word „everything“.

Restrictions and Priorities

Restrictions are used to introduce look-ahead for productions. Most commonly, they are used to enforce the longest-match policy on identifiers.

```
1 lexical syntax
2     IDENT = [a-zA-Z0-9\_]+
3 lexical restrictions
4     IDENT -\[a-zA-Z0-9\_]
```

Snippet 5.4: Identifier definition and "longest-match" restrictions

Priorities are used to define more complex disambiguation conditions. They use their own section, as can be seen in the example below, and multiple priorities in one section are divided by comma. Priorities are mathematically transitive.

```
1 context-free priorities
2     {Exp.Times} >
3     {Exp.Plus Exp.Minus},
4     {Exp.Plus Exp.Minus} >
5     {Exp.Equals}
```

Snippet 5.5: Priorities example from <http://metaborg.org/sdf3/>

In this example, the `Exp.Times` production always supersedes the `Exp.Plus` and `Exp.Minus` productions, which are on the same priority level. Also, all three are prioritized over `Exp.Equals`.

Layout Characters

are defined by the reserved `LAYOUT` production. Characters defined as `LAYOUT` will be ignored if encountered between productions or parts of a production, but not if they appear inside a literal. There they are still required. Usually `LAYOUT` consists of whitespace characters and has a longest-match policy enforced.

Templates

ease the use of the completion and pretty printer auto generation as they pre-define what the language should look like. They almost look like regular context-free productions with the only difference, that the left-hand side is enclosed in pointy brackets. Additionally productions that appear on the left-hand side can be enclosed in pointy or square brackets to act as placeholders. This syntax is equal to the completion template syntax described in section 6.1.3 but has the advantage that the user can define which symbols are allowed or expected between parts of the template and its end more easily via Template Options.

5.2 The Name Binding Language (NaBL)

Spoken „enable“ [11], this language provides the name definition and analyzing support for the DSLs designed in Spoofax. Specifically it offers functionality for namespaces, name declaration and references as well as scopes. Additionally it allows to import NaBL name binding rules from other languages to allow multi-layer language references. Besides the specification on metaborg.org/nabl/, section 6.1.4, beginning at page 49, offers an elaborate description with examples from the ExaSlang Layer 4 implementation.

5.3 The Type Specification Language (TS)

TS is complementary to NaBL as it tackles the type bindings and analysis of Spoofax languages through a declarative approach. TS files are currently not pregenerated in Spoofax and therefore have to be added and included by the user manually to the `\trans` folder.

```
1 module types
2
3 type rules // binding
4
5   Var(x) : t
6   where definition of x : t
```

Snippet 5.6: Type specification example from metaborg.org/ts/

The example in listing 5.6 from the Type Specification Language homepage metaborg.org/ts/ shows the beginning of a type module in Spoofax. A simple binding rule assigns the type `t` to the variable `x` when the definition of `x` matches `t`. This can be extended to make complete expression analysis, as is exemplarily described at metaborg.org/ts/. As there are sections of the TS reference page that are incomplete, this might suggest that TS is not feature complete yet. Note that type analysis is not implemented for the ExaSlang IDE so far and it is unknown to the author if it works properly in the current version of Spoofax at all.

5.4 The Spoofax Testing Language (SPT)

SPT is, like TS, not realized in the ExaSlang IDE yet and holds the same restrictions regarding its functionality. But, in difference to TS it is wildly feature complete and fully described at metaborg.org/spt/. It allows automated tests to examine the consistency and functionality of „(...) parsing rules, operator precedence and associativity, the abstract syntax tree, errors and warnings, and transformations (...)“ [11].

```
1 module Expression-tests
2
3 start symbol Exp
4
5 test Add [[ 1 + 2 ]] parse succeeds
6
7 test Multiply has correct AST [[ 1 * 2 ]] parse to Mul(Int("1"),_)
8
9 test Parentheses [[ (1 + 2) ]]
10
11 test Multiply and add precedence [[ 1 * 2 + 3 ]]
12   parse to [[ (1 * 2) + 3 ]]
```

Snippet 5.7: Language testing example from metaborg.org/spt/

The tests in this example, listing 5.7, begin at the start symbol `Exp`. They check if the parse for `1+2` in the `Add` production does succeed and examine if `1*2` results in the AST `Mul(Int("1"),_)`. With the second argument being an unspecified placeholder, the parentheses test checks whether the expression `(1 + 2)` is valid. But it is unclear if there is any effect on the end result, as there is no success or failure condition specified. In the last test a simplified version of the AST parsing is used. It simply checks if `1 * 2 + 3` parses to the same AST-node as `(1 * 2) + 3`. For further instructions on how to write SPT refer to metaborg.org/spt/.

5.5 The Editor Services Language (ESV)

Although not explicitly mentioned on MetaBorg as a stand-alone language, the Editor Services Language does qualify as one. It is a conglomerate of small defining sections each describing the contents of one specific editor service feature. These sections are directly interpreted by Spoofox. In addition to the description given in this chapter, <http://metaborg.org/spoofox/editor-services/> offers a complete overview and additional examples and explanations. The overall nine editor services, or features, can be divided in three groups. First there are those completely defined in ESV, second those only offering basic interface information to the actual feature implementation, usually done in Stratego, and third services that are relatively static and define some basic properties of the Editor and the Language.

The first group consists of Foldings, Completions and Colorer. Those three are presented in detail in section 6.1 at the example of ExaSlang Layer 4.

In the second group one can find Views, References and Refactorings. Menus, Syntax from the third and Main from the last group.

Views are by default only providing an interface to the two non editor windows used by Spoofox Editors, the outline view, shown in figure 3.13 on page 13 and the properties view shown in figure 5.1. The Views code shown in listing 5.8 shows the whole, unchanged, views file of Exa4 as it has been generated. Both views point to Stratego functions that then calls into Java functions that link into the Eclipse UI environment. Technically it is possible to add further views by implementing them in Spoofox and then linking them to this service. Nonetheless, it is often easier to directly implement GUI addons directly in Java together with the Eclipse interface for extensions, explained in section 6.2.

```
1 module Exa4-Views
2
3 views
4
5   outline view: editor-outline (source)
6     expand to level: 3
7
8   properties view: editor-properties
```

Snippet 5.8: The Exa4-Views.esv file unchanged from its state after initial generation

Reference simply lists all Stratego strategies used for the reference and type system and does not need to be edited by the programmer at all. For the actual implementation of references see sections 5.2 and 6.1.4.

Refactoring only defines the UI information as shown in listing 5.9 from metaborg.org/spoofox/tour/ in the Refactoring Specifications section.

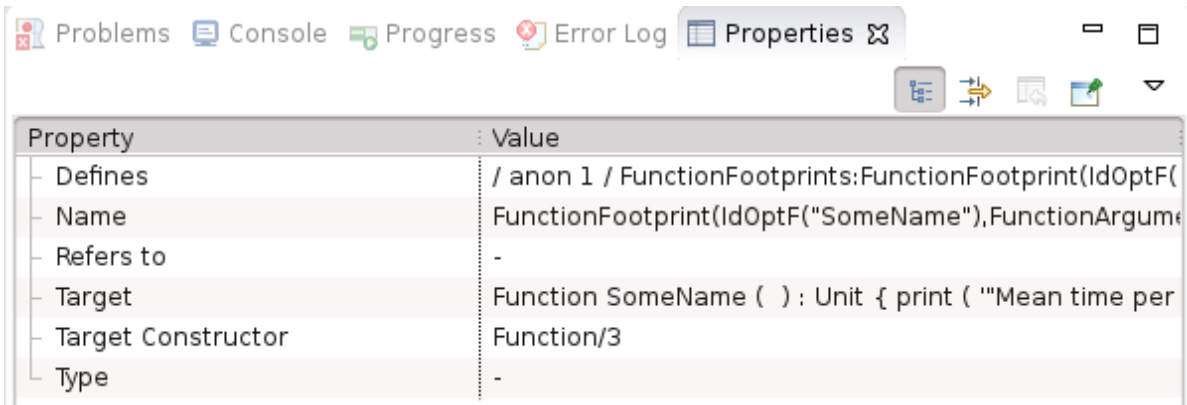


Figure 5.1: Spoofax properties view showing the AST properties of the main node of a short .exa4 file

```

1 refactorings
2   refactoring ID: "Rename" = rename(cursor)
3   shortcut: "org.eclipse.jdt.ui.edit.text.java.rename.element"
4   Input
5     identifier: "New name"=""

```

Snippet 5.9: Refactoring example from metaborg.org/spoofax/tour/

It results in a window with an empty field asking for a new name. As the actual refactoring is a transformation, not surprisingly it is written directly in Stratego. The corresponding file can be found at `\trans\refractor.str` in the Spoofax project. Further details on the layout of refactoring strategies can be found at metaborg.org/spoofax/tour/ under „Refactoring Transformations“.

Menus are used to directly link Stratego functions to the GUI. It generates an arbitrary amount of menus and submenus as buttons in the Eclipse toolbar. Each can contain several commands, called actions, which specify Stratego function calls. Additional keywords specify the behavior of the specific menu or action. An example can be seen in listing 5.10.

```

1 menus
2   menu: "Analysis"                                (meta)
3
4   action: "Reset and reanalyze"                    = debug-reanalyze
5
6   submenu: "Show analysis"                          (openeditor)
7     action: "Project"                              = debug-project
8     action: "Partition"                            = debug-partition
9   end

```

Snippet 5.10: Menus example

Syntax derives its contents from the default LAYOUT definitions of Spoofax languages, and does not adjust to changes made by the DSL programmer. It provides the editor with information about when to do indents automatically or what literals function as fences. For whitespace unaware languages, the default behavior usually is good enough to work properly. This is especially the case when deviating language constructs, like the coloring of identifiers, are already handled in other parts of the editor specifications. Therefore it is usually unnecessary to make changes to this part of the editor specification.

```
1 module Exa4-Syntax.generated
2
3 language Syntax properties (static defaults)
4
5   // Comment constructs:
6   line comment           : "//"
7   block comment          : "/*" * "*/"
8
9   // Fences (used for matching,
10  // inserting, indenting brackets):
11  fences                  : [ ]
12                           ( )
13                           { }
```

Snippet 5.11: Snippet from a default syntax file

Main usually only needs to be edited if some basic changes to the language are made, for example the associated file extension or the name of the language itself is changed. Otherwise it just merges all of the other ESV files together in one place by importing them. Furthermore it offers some meta information, like a general description and an url that may appear in specific Eclipse queries.

```
1 language General properties
2
3   name:      Exa4
4   id:        exa4
5   extends:   Root
6
7   description: "Spoofax-generated editor for the ExaSlang Level4
8               language"
9   url:        http://www.exastencils.org/
10
11  extensions: exa4
12  table:      include/Exa4.tbl
13  start symbols: Start
```

Snippet 5.12: Snippet from the unchanged main file of Exa4

5.6 The Stratego Transformation Language

```
1 rules // Editor services
2
3 // Resolves a reference when the user control-clicks or presses
  Shift-F3 in the editor.
4 editor-resolve:
5     (node, position, ast, path, project-path) -> definition
6     where
7         definition := <analysis-resolve(|<language>, project-path)>
           node
```

Snippet 5.13: A Stratego function used for languages defined in Spoofox to enable the resolving of references and locating their position in the code,

Most languages used in Spoofox, directly or indirectly, are transformed into Stratego during code generation or at least do reference it strongly. Because Stratego was a stand-alone project before being merged with Spoofox and later the MetaBorg project its documentation is still in transfer to the MetaBorg homepage⁵ and can currently be found in the Stratego documentation⁶. Unfortunately the important manual and tutorial paths can, as of 2015/03/30, no longer be accessed making it difficult to work with at the moment. Thankfully this is not that great of a problem, because there are only three Spoofox features requiring to be written directly in Stratego. These are the outliner and the pretty printer as well as the transformations of DSL code into general languages like Java. The latter was not scope and goal of this thesis anyway. A „how to“ description of the outliner functionality can be found in section 6.1.2, as it is already implemented in the ExaSlang IDE. Figure 3.13 on page 13 shows an outliner generated by the Spoofox outline function.

Not realized during the implementation of the ExaSlang IDE was the pretty printer functionality. The pregenerated functions were buggy, as they did not add any line breaks to their output, as can be seen in figure 5.2. It was therefore deactivated in the end user version. This is clearly a bug in the current version of Spoofox and a reintroduction of this feature to a later point is quite likely.

⁵<http://metaborg.org/stratego/>

⁶<http://strategoxt.org/Stratego/StrategoDocumentation>

```

Domain global < [ 0 , 0 , 0 ] to [ 1 , 1 , 1 ] > Layout NoComm < Real , node > @
all { ghostLayers = [ 0 , 0 , 0 ] duplicateLayers = [ 1 , 1 , 1 ] } Layout
CommPartTempBlockable < Real , Node > @ all { ghostLayers = [ 0 , 0 , 0 ]
duplicateLayers = [ 1 , 1 , 1 ] } Layout BasicComm < Real , Cell > @ all {
ghostLayers = [ 1 , 1 , 1 ] with communication duplicateLayers = [ 1 , 1 , 1 ] with
communication } Layout CommFullTempBlockable < Real , Cell > @ all { ghostLayers =
[ 1 , 1 , 1 ] with communication duplicateLayers = [ 1 , 1 , 1 ] with communication
} Field Solution < global , BasicComm , 0.0 > [ 2 ] @ ( coarsest to 0 ) Field
Solution < global , CommFullTempBlockable , 0.0 > [ 2 ] @ ( 1 to finest ) Field
Residual < global , BasicComm , None > @ all Field RHS < global , NoComm , None > @
( coarsest to 0 ) Field RHS < global , CommPartTempBlockable , None > @ ( 1 to
finest ) Field VecP < global , BasicComm , None > @ coarsest Field VecGradP <
global , NoComm , None > @ coarsest Stencil Laplace @ all { [ 0 , 0 , 0 ] => 6.0 [
1 , 0 , 0 ] => -1.0 [ -1 , 0 , 0 ] => -1.0 [ 0 , 1 , 0 ] => -1.0 [ 0 , -1 , 0 ] =>
-1.0 [ 0 , 0 , 1 ] => -1.0 [ 0 , 0 , -1 ] => -1.0 } Stencil CorrectionStencil @ all
{ [ 0 , 0 , 0 ] => 0.0625 [ x % 2 , 0 , 0 ] => 0.0625 [ 0 , y % 2 , 0 ] => 0.0625
[ x % 2 , y % 2 , 0 ] => 0.0625 [ 0 , 0 , z % 2 ] => 0.0625 [ x % 2 , 0 , z % 2 ]
=> 0.0625 [ 0 , y % 2 , z % 2 ] => 0.0625 [ x % 2 , y % 2 , z % 2 ] =>
0.0625 } Stencil RestrictionStencil @ all { [ 0 , 0 , 0 ] => 1.0 [ 0 , 0 , -1 ] =>
0.5 [ 0 , 0 , 1 ] => 0.5 [ 0 , -1 , 0 ] => 0.5 [ 0 , 1 , 0 ] => 0.5 [ -1 , 0 , 0 ]
=> 0.5 [ 1 , 0 , 0 ] => 0.5 [ 0 , -1 , 1 ] => 0.25 [ 0 , -1 , -1 ] => 0.25 [ 0 , 1
, 1 ] => 0.25 [ 0 , 1 , -1 ] => 0.25 [ -1 , 0 , 1 ] => 0.25 [ -1 , 0 , -1 ] => 0.25
[ 1 , 0 , 1 ] => 0.25 [ 1 , 0 , -1 ] => 0.25 [ -1 , -1 , 0 ] => 0.25 [ -1 , 1 , 0 ]
=> 0.25 [ 1 , -1 , 0 ] => 0.25 [ 1 , 1 , 0 ] => 0.25 [ -1 , -1 , 1 ] => 0.125 [ -1
, -1 , -1 ] => 0.125 [ -1 , 1 , 1 ] => 0.125 [ -1 , 1 , -1 ] => 0.125 [ 1 , -1 , 1 ]
=> 0.125 [ 1 , -1 , -1 ] => 0.125 [ 1 , 1 , 1 ] => 0.125 [ 1 , 1 , -1 ] => 0.125
} Globals { } Function VCycle @ coarsest ( ) : Unit { VCycle_0 @ current ( ) }
Function VCycle_0 @ coarsest ( ) : Unit { UpResidual @ current ( ) communicate
Residual @ current Variable res : Real = NormResidual_0 @ current ( ) Variable
initialRes : Real = res loop over VecP @ current { VecP @ current = Residual @
current } Variable cgSteps : Integer repeat 512 times count cgSteps { communicate
VecP @ current loop over VecP @ current { VecGradP @ current = Laplace @
current * VecP @ current } Variable alphaDenom : Real = 0 loop over VecP @ current
where x > 0 && y > 0 && z > 0 with reduction ( + : alphaDenom ) { alphaDenom
+= VecP @ current * VecGradP @ current } Variable alpha : Real = res * res /
alphaDenom loop over Solution @ current { Solution [ curSlot ] @ current +=
alpha * VecP @ current Residual @ current -= alpha * VecGradP @ current }

```

Figure 5.2: Exemplary Exa4 file converted by Spoofox generated pretty printer, shown in KATE with automatic line wrapping

6 ExaSlang in Spoofax and Eclipse

6.1 Spoofax Features

Remarks

While Spoofax offers a wide variety of automated feature generation the implementation for ExaSlang has shown, that these mechanisms are not yet good enough to satisfactorily generate the front-end features used in an editor. This leads to a considerable effort after implementing the language itself, to adjust, expand and correct the pregenerated features in a way that results in a good end user experience. To support future work on ExaSlang, this section describes the implemented features in detail and gives hints on how to expand and adjust them in the future. Note that the original SDF3 implementation of the languages, especially for ExaSlang Layer 4, while formally correct, has been proven incapable of satisfying the needs of the various editor features. This led to the insertion of additional transformation steps and the separation of syntactically similar productions into separate ones, that now have to be maintained on their own as well. Overall this led to an about 20 % extended code base for the language definition and certainly will lead to additional maintenance effort in the future.

```
1 Function.Function = FUNCTION IdentWithOptLevel
2   "(" FunctionArgumentList ")" ":" ReturnDatatype "{" FunctionBody "}"
3 VariableDeclaration.VariableDeclaration = VARIABLE
4   IdentWithOptLevel ":" Datatype
5
6 IdentWithOptLevel.IdOpt_Level = IDENT Level
```

Snippet 6.1: Original Exa4 SDF3 definition

```
1 Function.Function = FUNCTION FunctionIdentWithOptLevel
2   "(" FunctionArgumentList ")" ":" ReturnDatatype "{" FunctionBody "}"
3 VariableDeclaration.VariableDeclaration = VARIABLE
4   VariableIdentWithOptLevel ":" Datatype
5
6 FunctionIdentWithOptLevel.IdOptF_Level = IDENT LevelTop
7 VariableIdentWithOptLevel.IdOptV_Level = IDENT Level
```

Snippet 6.2: Extended Exa4 SDF3 definition

The comparison between listing 6.1 and listing 6.2 shows that the original `IdentWithOptLevel.IdOpt_Level` production has been split into `FunctionIdentWithOptLevel.IdOptF_Level` and

`VariableIdentWithOptLevel.IdOptV_Level`.

This has been done to make them distinguishable for the reference resolver. Without doing this, there would be no difference when defining the name of a Function to a Variable from the NaBL point of view. Note, that this example is a slightly changed and shortened version of the original code used for the Exa4 IDE and is purely meant to illustrate the problem described before.

Manual

This chapter also serves as manual and reference for future contributors to the IDE. It is written in such a way that new contributors are able to orientate themselves and learn how and where they can make changes and additions to the existing features of the project.

Version

The document is based on the Version 1.1.8.release of the ExaSlang feature, deployed on April 4, 2015.

6.1.1 Syntax Highlighting

The ExaSlang editor uses three parts of the Spoofax coloring scheme. The source files are located in `\editor\<LANGUAGE>-Colorer.esv`. An Example of the Syntax Highlighting offered by Spoofax can be found on page 8 in figure 3.3.

Lexical Coloring is the simplest part, all default syntax components are directly colored via the scheme:

`keyword : Color`

Particularly all numerals, integers and reals alike, are subject to this. It will apply when no other coloring schemes supersede it and the parser correctly identifies those components, which is not guaranteed in deep AST-branches.

Syntactical Coloring is somewhat more complex and is split into two parts. The first one, following the scheme `SORT.CONSTRUCTOR : Color` colors all literals of the specified constructs. Sub-constructs keep their own colors, or, if not specifically colored, will use the default ones. To override sub-construct schemes, the keyword „environment“ is used which will force the entire construct including all sub-constructs, but not sub-sub-constructs, to use the uppermost coloring scheme. As always when the `SORT.CONSTRUCTOR` scheme is used either of them can be substituted by a `_` to define all possible sorts or constructors at this place.

`environment SORT.CONSTRUCTOR : Color`

A color is either directly assigned by specifying the three 8bit unsigned integer RGB values, optionally followed by the **bold** and/or *italic* keywords. Alternatively a predefined color can be named. But those two methods can not be mixed.

```

1 AccIntVar.AAccIntVar : 0 0 0 italic
2 AccIntVar.AAccIntNum : color_numbers
3
4 At._ : 128 0 0 bold
5 Level._ : _ bold
6 environment Level._ : _ bold

```

Snippet 6.3: Example for color assignment to productions

The definition of a color follows the same pattern as its assignment, but replaces the colon with a "=" and looks the way shown in listing 6.4.

```

1 color_numbers = 245 140 0 //darkorange
2 color_index = 0 0 128 // darkblue
3 color_level = 128 0 0 bold // dark red // color_keyword
4 color_fctcall = 0 0 0 italic // black

```

Snippet 6.4: Definition of colors

It is also possible to assign an additional name to an existing color definition, as has been done in listing 6.5.

```

1 gray = 128 128 128
2 grey = gray

```

Snippet 6.5: redefining color names

A placeholder `_` can be used to add additional style tags to an assignment or color definition without overwriting the previous color settings.

```
environment Level._ : _ bold
```

6.1.2 Outliner

In difference to the other implemented Spoofox editor features, the outline is directly written in Stratego and can be found in the `<LANGUAGENAME>-outliner.str` file in the `\editor` folder.

To generate an outline term, the predeclared Stratego rule `to-outline-label` is used, which is called by the Spoofox runtime every time a constructor is invoked. The given constructor then is transformed by an arbitrary amount of Stratego rules and the final result listed as a string in the outline tree, that follows the AST branching. Shown in figure 3.13 on page 13 is a fully fledged Exa4 outline view.

```
to-outline-label : Def_Globals(_) -> "Globals"
```

This simple example of such a transformation is called when a Global clause is opened in ExaSlang Layer 4. For the outline function it transforms the constructor `Def_Globals(_)` to the string "Globals". This string then is used as a node in the outline view as can be seen in figure 6.1.

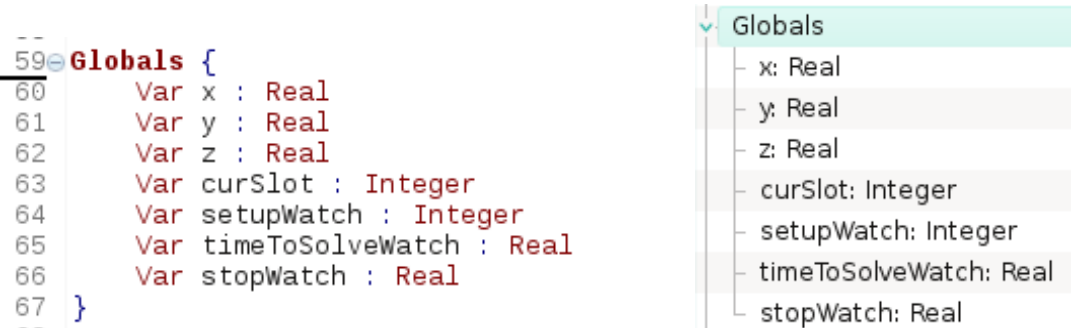


Figure 6.1: A Global structure with several variable defines and its resulting outliner

```

1 to-outline-label = ?Domain(_,name,_,_);
2                 !("Domain: \"",Name,"\"");
3                 conc-strings

```

Snippet 6.6: Generation of an outline string

This invocation of `to-outline-label` uses the strategy `syntax-style` instead of the simple `rule-syntax`. The transformation from the constructor to the resulting outline nodes has three steps.

The first step is to analyze whether the right constructor is present at the current Abstract Syntax Tree-Node, by specifying the constructor and using the leading "?" literal. When this check is passed the "!" literal denotes the creation of a new term. `!("Domain: \"",Name,"\"")` creates the string „Domain: "Name" “, with `Name` being the content of the variable `name` defined in the constructor. Note that the constructor actually has four variables, but as only the second is used the others are ignored by using the "_" literal as a placeholder.

As the variables are actually AST-Nodes themselves, in the last step, the rule invocation `conc-strings` removes all literals from the AST-Node to string transformation and leaves only the actual, human readable, variable content of the node. In this case the string specified before with the content of the IDENT is extracted from `Name`.

If the actual information is deeper below in the AST it has to be extracted by sub-transformations that convey it to the AST-Level used in the outline label call. This can be necessary if the unique node marking an object that should be outlines has sub-constructors in common with other nodes that have different or no outlines at all. This is the case with the listing of global variables of the Global scope used as example above.

```
to-outline-label = ?GlobalEntry_Variable(<varinfoextraction>)
```

Instead of simply applying a name to the variables another strategy is called that has the AST-Node serving as variable base. So it is one level beneath the initial node. Syntactically the strategy name is surrounded by pointy brackets. The result of such a call can be assigned to a variable by using `=> Variable-Name` after the call name inside the brackets, as can be seen in the following example.

```
to-outline-label = ?FieldLevel(<level>extraction => Level>); !(Level)
```

Here the level is extracted from the sub-nodes, assigned to the variable level and then the result is applied to the initial rule, overwriting its previous contents.

In case there are several different possible constructors appearing in the same place inside a node, the analyzing sub-rule can use the `or()` case to check which of the possibilities is true and then continue from there on.

```

1 level>extractionB =
2   or (
3       ?LevelType_Relative(<level>extractionR => text>),
4       or (
5           ?LevelType_Range(<level>extractionP => text>),
6           or (
7               ?LevelType_List(<level>extractionL => text>),
8               or (
9                   ?LevelType_Simple(<level>extractionS => text>),
10                  ?LevelType_Negation(NegLevel(<level>extractionN => text>))
11              )
12          )
13      )
14  );!(text)

```

Snippet 6.7: Extraction of information from deeper nodes in the AST

The `or()` case can only handle two entries at a time and therefore must be nested with other `or()` cases if more than two options are possible. Syntactically the first option is always separated by a comma `,` from the second option. The `"?"` literal is also needed to actually analyze something inside of an option clause, otherwise it would always default to true. As the analysis is done before the strategy call, the above construct only assigns a value to text if the analysis was successful. Note that it is not specified what happens if several options result in valid cases.

```
funcname>extraction : IdOptF(Name) -> Name
```

If only a direct transformation of a constructor variable to a string is necessary, the rule-syntax can be used. The above rule returns the same result as the strategy below would do.

```
funcname>extraction = ?dOptF(Name);!(Name);conc-strings
```

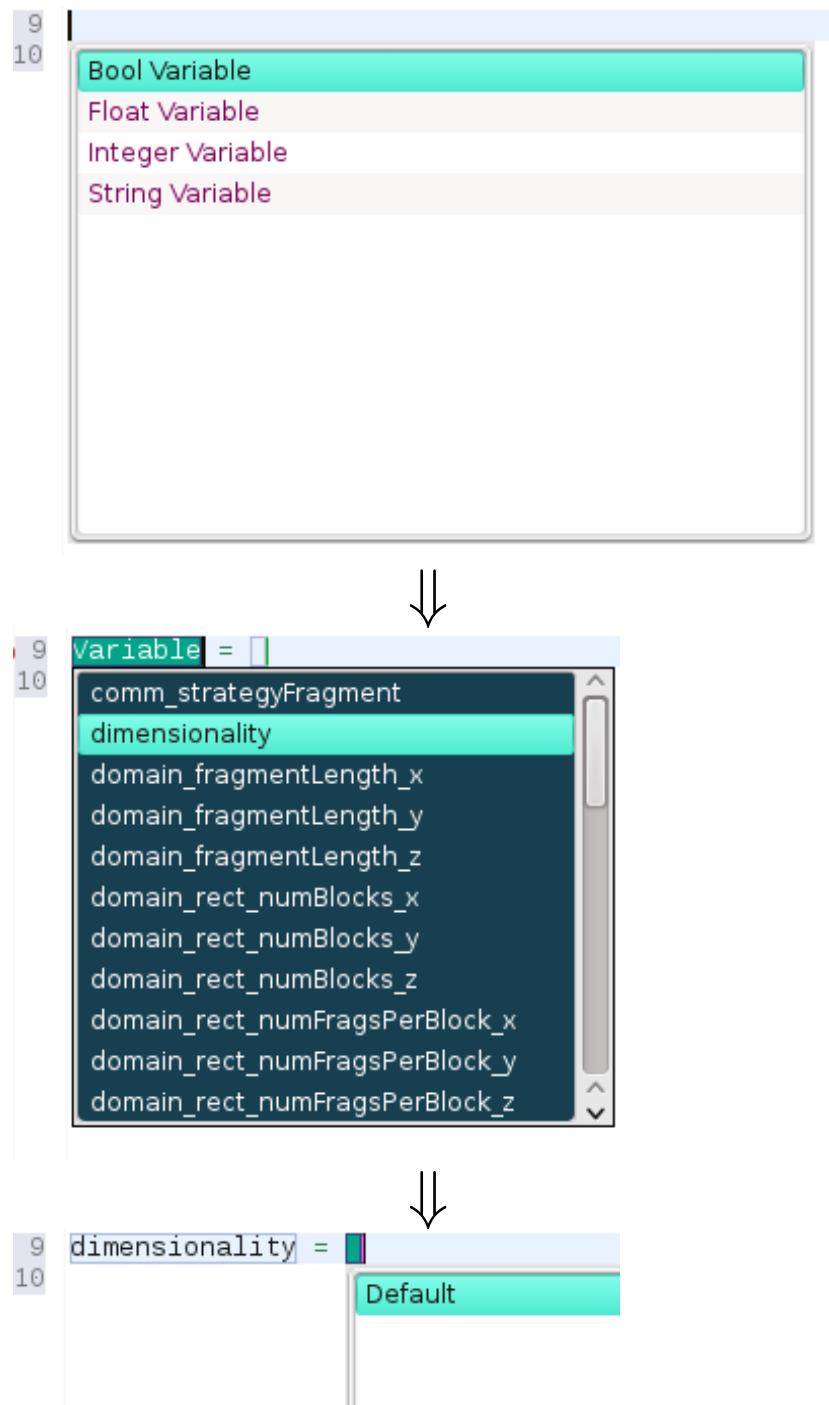


Figure 6.2: Knowledge Completion Menu - three steps to a complete entry

6.1.3 Completions

Implementation in ExaSlang Technically Spoofax supports both syntactical and semantic completion, however, the current version of the ExaSlang IDE only supports the first one. The code can be found in the `\editor` folder in the `<LanguageName>-Completions.esv` file. The style of the completion templates is different in ExaSlang Layer 4 as well as each of the two configuration languages. Exa4 concentrates on offering templates to almost all productions and keywords that can be accessed where syntactically allowed, which is very similar to the standard Java completions in Eclipse. Exa-Settings simply offers a list of all keywords which result in a template that already is correctly formatted and only has to be filled with a valid value.

Exa-Knowledge, with its more diverse values and the larger amount of keywords, offers multi-step assistance to find correct combinations, which is shown in figure 6.2. The user first can choose between different data types, and then between the actual keywords eligible for the chosen type. Then the final step offers the choice between the default value for the chosen keyword, a permitted custom value, or, in case of variables with only a small amount of allowed values, a list of all possible values. Note that the editor does not check whether certain combinations of different values in the knowledge file are valid, because the NaBL does not support non-literal or dynamic associations. Checking for those connections could appear when type detecting is implemented, as similar analyses are part of the TS system. Still, it is not supported natively by the Spoofax environment and it could turn out to require the implementation of a custom Stratego or Java functionality or not be possible to be properly included in the editor at all.

The Exa4 completions work in a similar fashion by offering lists of allowed productions at the current cursor position. They have variables which themselves offer either lists with sub-productions or keywords accepted at their position to the user. Because of the size of some production trees, the offered lists usually are not in line with the actual language definition, but offer choices only from tree nodes that are actually relevant to the user.

Coding Details In detail the `completions` section is divided in `completion template` and `completion trigger` structures. The templates specify a production of any kind where it should be triggerable followed by the textual representation of the template in the completion list and finished with the actual template contents. The last part is optional and when no content is specified, the textual representation will be used instead.

```
1 completion template TYPE : "Unit"
2 completion template TYPE : "Array"
3 completion template TYPE : "Real"
4 completion template TYPE : "Integer"
5 completion template TYPE : "String"
6 completion template TYPE : "Complex"
```

Snippet 6.8: Templates for keywords

This code snippet shows a simple completion template for the `TYPE` lexicals that result in a list of all types available. As can be seen, completion templates do not have to be unique. Multiple templates for the same production simply result in a larger list of choices for the user.

```
1 //Function Definition Templates
2 completion template Function :
3   "Function Name@Level ( ) : Returntype { }" =
4   "Function " <Name@Level ( Arguments ) Returntype:FunctionFootprint> "
5     { }" (blank)
6
7 completion template FunctionFootprint :
8   "Name@Level ( ) : Returntype" =
9   <FunctionIdentWithOptLevel:FunctionIdentWithOptLevel> " ( "
10     <Arguments:FunctionArgument> " ) : " <Returntype:ReturnDatatype>
11
12 completion template FunctionFootprint :
13   "Name@Level ( ) : Returntype" =
14   <FunctionIdentWithOptLevel:FunctionIdentWithOptLevel> " ( "
15     <Arguments:FunctionArgument> " ) : " <Returntype:ReturnDatatype>
16
17 completion template FunctionArgument :
18   "Name : Datatype" = <Name:IDENT> " : " <Datatype:Datatype>
```

Snippet 6.9: Examples for completion templates with multiple levels

The more complex templates above are used for the function definition production. The string between the ":" and the "=" literals is an one-to-one representation of the text later shown in the completion menu. After the equals sign the completion consists of strings that will literally translate into the inserted completion and the "< >" expressions, marking the section as a variable that uses more templates to be completed. The first part of the pointy bracket expression is the string representation of the variable in the template and works similarly to pure strings.

In the second part, after the colon, another production specifies which template should be called for further completion. This production does not actually have to be valid at the current position of the AST from a language point of view but needs to have a completion template defined. As can be seen in the code example, this functionality can be nested to allow complex guided templates for almost all imaginable language constructs. By using the `(blank)` property keyword the editor only will offer the completions in empty lines, a helpful tool to avoid bad syntax and keep the completion list from overcrowding. To navigate a template Spoofox uses the same mechanisms Eclipse uses for Java and other language completions. Pressing `alt-space` presents the user, if possible, with a list of templates. When the template has been chosen and confirmed

by pressing **enter**, the editor enters into the template editing state. There the user can navigate between the variables of the template by pressing **tabulator** and calling lists of sub-templates by pressing **alt-space**. Note, however, that once entering a sub-template the editor closes the old template environment without the possibility to reopen it. So entering a sub-template prohibits from regularly entering another of the sub-templates. This behavior is not Spoofax exclusive, but due to the complex nature of ExaSlang Layer 4 it is quite noticeable here compared to generic languages like Java.

6.1.4 Reference and Name Resolving

For reference and name resolving in ExaSlang the Name Binding Language, mentioned already in section 5.2, is used. While the other editor feature source files can be found in the `\editor` folder, the file `names.nab` is located in the `\trans` folder. The Spoofax developing environment translates the file during runtime into the `names.str` Stratego file in the same folder. The reference files in the `\editor` folder only serve as a link to the analyzer functions of Stratego and, although they can be customized, they have not been changed for the ExaSlang implementation. An example how reference errors look like in the resulting editor is given on page 11 in figure 3.10.

Like many of the editor features, the nabl language uses constructors and their variables as base for its functionality. Like at other places in Spoofax, it is not advisable to use a constructor more than once inside a given section, due to undefined behavior of double occurrences. This is especially important for the reference and name resolving definitions, as they consist basically of three different parts using the same constructor calls. Additionally a list of all used namespaces must be defined, beforehand. The one for Exa4 can be seen in the code snippet below.

```
1 namespaces
2   FunctionNames
3   FunctionFootprints
4   StencilNames
5   StencilFootprints
6   StencilFieldNames
7   StencilFieldFootprints
8   FieldNames
9   FieldFootprints
10  LayoutNames
11  LayoutFootprints
12  VariableNames
13  VariableFootprints
14  GlobalVariableNames
15  DomainNames
```

Snippet 6.10: List of all namespaces in Exa4

These will be referenced by the **binding rules** section to link different objects together in their corresponding namespaces. For example, all functions will hand their name into the **FunctionNames** namespace and the complete function footprint into the **FunctionFootprints** namespace. A similar pattern is used for all other types of objects. Note that in Exa4 there exist two **VariableNames** namespaces, one for local variables defined inside a function, and one for global variables defined implicitly by the language itself, or in the Global scope by the user. This has to be done due to namespace scoping.

```

1 binding rules
2   Programm (_):
3     scopes
4       FunctionNames, FunctionFootprints,
5       StencilNames, StencilFootprints,
6       StencilFieldNames, StencilFieldFootprints,
7       FieldNames, FieldFootprints,
8       LayoutNames, LayoutFootprints,
9       VariableNames, VariableFootprints,
10      GlobalVariableNames,
11      DomainNames

```

Snippet 6.11: Scoping example

This example shows all namespaces, that are scoped inside the **Programm(_)** constructor. Currently only variables have local scopes in ExaSlang. These are scoped inside of functions, including the function parameters, and inside all predefined bodies of loops and conditional terms.

```

1 Function (_,Footprint,_):
2   defines FunctionFootprints Footprint
3   scopes VariableFootprints, VariableNames
4
5   IdOptF_Level (Name,_):
6     defines non-unique FunctionNames Name
7   IdOptF (Name):
8     defines non-unique FunctionNames Name
9
10  FunctionArgumentList(Footprint):
11    defines VariableFootprints Footprint
12
13  FunctionArgument(Name,_):
14    defines non-unique VariableNames Name

```

Snippet 6.12: Variable name definitions

Every object declaration defines a unique footprint, consisting of its name and parameters like level, types and in case of functions variables, that identify them as a whole and may not occur twice in the same section of code. Furthermore, to be able to

check whether an object call is valid, their identifiers are stored separately and with the **non-unique** keyword. This allows multiple objects of the same type to have the same name and only be separated by their actual footprint.

```

1 FunctionCall_Leveled (Name,_,_):
2   refers to FunctionNames Name
3 FunctionCall_Flat (Name, _):
4   refers to FunctionNames Name

```

Snippet 6.13: Variable name resolving

Object access is then checked as in the code above with the **refers to NAMESPACE** **Variable** phrase. In cases where a specified reference may refer to multiple namespaces the second and all following refer-to phrases are led by the **otherwise** keyword. The example below shows this for the assignment construct, once with and once without an additional level access. Note, that any unique constructor may only be called once in all of the type binding modules. This means, that it has to encapsulate all parts of the name binding and different modules can not extend or modify this call later on.

```

1 Assignment_FieldLike (Name,_,_):
2   refers to VariableNames Name
3   otherwise refers to GlobalVariableNames Name
4   otherwise refers to StencilNames Name
5   otherwise refers to FieldNames Name
6
7 Assignment_Flat (Name, _):
8   refers to VariableNames Name
9   otherwise refers to GlobalVariableNames Name

```

Snippet 6.14: Multiple choice name resolving

In the name binding file of Exa4 there is also a list of implicitly prebound functions and variables, for example mathematical functions like `sin`, `cos` and global variables like `X`, `Y`, `Z` coordinates of the current position in the domain or field. They are defined at the `Program._` scope and marked with the **implicitly** keyword, that allows to define non variable strings as references.

```

1 implicitly defines non-unique GlobalVariableNames
   "geometricCoordinate\_x"

```

Snippet 6.15: NaBL: Definition of internal variables

These references bind to internal functions of the languages and include access to a prefabricated timer framework, mathematical functions from the C Math.h library ¹ and corresponding mathematical constants. Additionally, a few other functions are provided, as well. Most prominently are a set of `print()` functions and a set of functions to invert stencils.

¹<http://www.cplusplus.com/reference/cmath/>

6.1.5 Folding

```
108 Function VCycle@((coarsest + 1) to finest) ( ) : Unit {
109     repeat 3 times {
110         Smoother@current ( )
111     }
112     UpResidual@current ( )
113     Restriction@current ( )
114     SetSolution@coarser ( 0 )
115     VCycle@coarser ( )
116     Correction@current ( )
117     repeat 3 times {
118         Smoother@current ( )
119     }
120 }
121
122 Function Smoother@((coarsest + 1) to finest) ( ) : Unit {
131 }
```

Figure 6.3: Expanded and folded function in a .exa4 file

As the folding generation of Spoofox is quite good, the generated files where only slightly changed for Exa4. Basically all top level declarations that have a body, marked by curly brackets, can be folded. The bodies of loops and conditionals clauses and longer listings, like a list of levels in an access declaration, can be folded as well. An example of how folding looks in Exa4 can be seen in figure 6.3. Settings and Knowledge neither have nor need any code folding.

```
1 folding
2   Layout._
3   Else.Else
```

Snippet 6.16: Basic folding examples

As folding is written in the .esv language of Spoofox, its syntax is fairly simple and straightforward. It consists of a single list, that contains either the sort, the constructor, or a valid combination of both and gives the resulting sub-AST the ability to be folded. \editor is the folder the source file <LanguageName>-Folding.esv can be found in.

6.2 Eclipse Features

Some of the features in the ExaSlang IDE are not indirectly provided by Spoofox but directly by functionalities in Eclipse itself.

Eclipse Plugins and Features The Eclipse environment is build in a way that makes it easy to expand and adjust the platform to the individual needs of the user. This is done via so called plugins, packages of Java code and linking information to other parts of

eclipse. Plugins, as single units of functionality expansions, are usually bundled together in features.

Plugins in the ExaSlang IDE Feature The ExaSlang IDE is such a feature, described in the previous section, consisting of one plugin for each individual language as well as a few additional plugins adding UI functionality to the IDE. Eclipse itself offers numerous options to ease the use of an external DSL. Only the ones used in the ExaSlang IDE project are listed here. These are project templates and UI extension. Project templates add a wizard that allows to create a new ExaSlang project in Eclipse that already is set up and configured properly to be used. The UI extension on the other hand integrates ExaSlang specific functions in the general GUI of Eclipse common to all distributions. Templates and UI extension serve as solid base for expansion of their functionality to meet future requirements. For this they use the extension feature for Eclipse plugins, whose interface can be seen in figure 6.4. It allows the plugin creator to easily link into the standard functionality of Eclipse. Specified in the plugin.xml file and supported by its own wizard page, the creator can choose from a list of possible extension points that are then added to the extension list of the plugin. These extension point entries have predefined fields that specify their name, internal IDs and other information, like icons for GUI elements, adjustable by the user. If the actual behavior of the extension can be customized inside the plugin itself, a Java class that contains the code for this behavior usually must be specified. If the class does not already exist the wizard supports its automatic generation with all needed sub-classes and basic functions. Afterwards, those then can be extended and changed by the creator. Additionally an

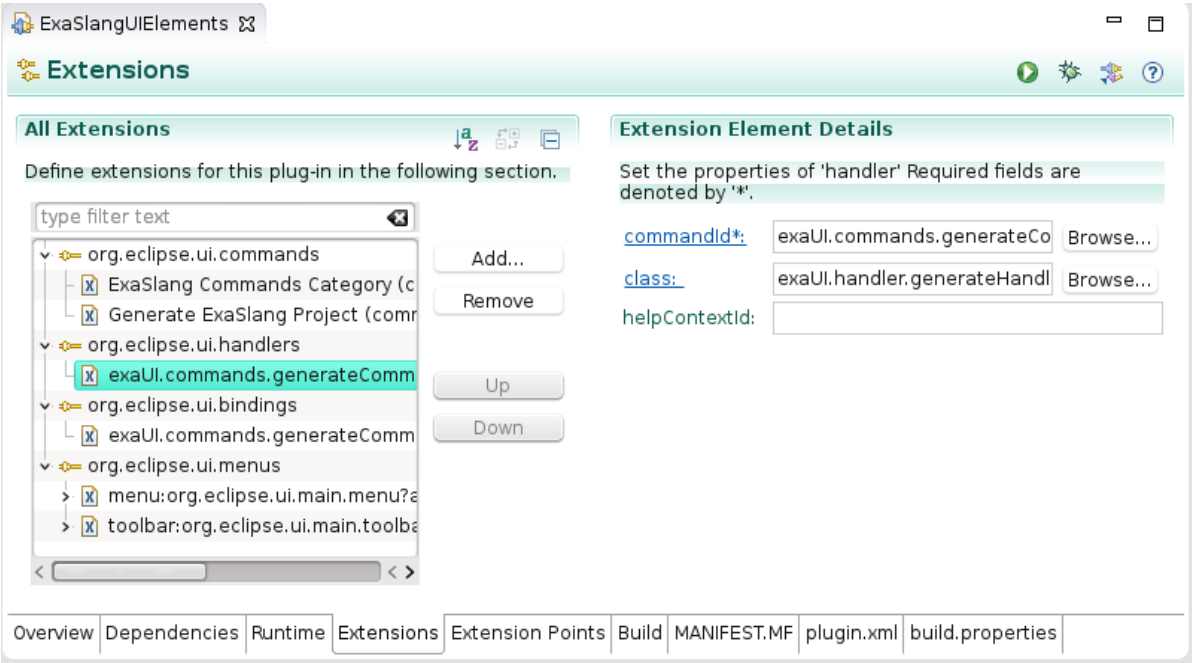


Figure 6.4: Extension wizard page of the ExaSlang UI project plugin.xml

update site project has been created to offer an easy and convenient way to deploy the IDE feature to the users via Eclipses own update mechanisms.

6.2.1 Project Templates

Templates, as introduced in section 3.3.1, are integrated into the ExaSlang IDE by using the Eclipse extension points for project wizards (`org.eclipse.ui.newWizards`). This approach allows direct integration with the Eclipse GUI and standard procedures as well as the ability to write and extend the feature in Java. Currently there is one project wizard implemented in the `newExaSlangWizard` package. It consists of two setup pages, the first one being the default page asking for project name and workspace location, and the second one asking for the location of the ExaSlang generator .jar archive. For details of the inner workings of this Eclipse extension please refer to the Eclipse homepage² and references³.

Upon completing the wizard, shown in figure 6.5 and figure 6.6, a new project with the folder structure and files specified in `ExaSlangSimpleProject.java` is created, the result can be seen in figure 3.14 on page page 14.

The ExaSlang template project is set up in a way, that new projects can be added easily. Besides having to add a new wizard extension, the user can simply create a new sub-class of the abstract `ExaSlangProject` class to initialize the creation of files and folders and add it to the pregenerated Wizard interface Java class. The derived `ExaSlangProject` class must implement the functions

`protected ArrayList<Folder>MakePaths(IProject project)` and `protected ArrayList<File>MakeFiles (IProject project)` which return an array-list of their respective types, either a File or a Folder definition. Note that File and Folder are inner classes of `ExaSlangProject` and not related to other File or Folder classes in Java. As the code snippet below shows, they simply consist of string variables containing their location, name and, in case of files, the content of the file itself. The method `calcPath()` can be used to return the complete path to the object.

```
1 public ExaSlangSimpleProject(String name, URI location, String genpath)
   {
2     super(name, location);
3
4     GeneratorPath = genpath;
5     BasePath = new String();
6     DSL = new Folder();
7
8     (...)
9
10    exa4 = new File();
11 }
```

²<http://www.eclipse.org/>

³<http://help.eclipse.org/luna/index.jsp>


```

12
13 protected ArrayList<Folder> MakePaths(IProject project) {
14     ArrayList<Folder> paths = new ArrayList<Folder>();
15     paths.add(DSL);
16     DSL.Name = "dsl";
17
18     (...)
19
20     return paths;
21 }
22
23 protected ArrayList<File> MakeFiles (IProject project) {
24     ArrayList<File> files = new ArrayList<File>();
25     String NEWLINE = System.getProperty("line.separator");
26
27     (...)
28
29     files.add(exa4);
30     exa4.Path = DSL.calcPath();
31     exa4.Name = "Source.exa4";
32     exa4.Text = "//Enter your Level 4 Code here" + NEWLINE
33               + NEWLINE;
34
35     (...)
36
37     return files;
38 }

```

Snippet 6.17: Definition of dsl folder and containing exa4-file

`ExaSlangSimpleProject` constructs its File and Folder information directly in the methods by creating new objects and directly writing the string information into them. More advanced approaches for complex needs are possible of course. To link the new project class to the actual wizard, in the wizard's Wizard class, the function `performFinish()` should create a new object of the project class and then call the `createProject()` function of the project class object.

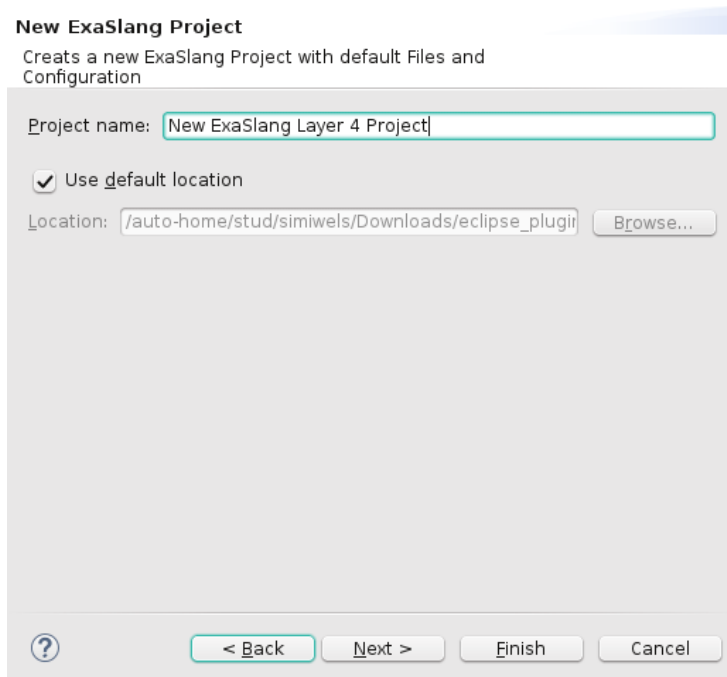


Figure 6.5: ExaSlang Project Wizard page one

6.2.2 UI Extension

Like the project template plugin, the UI plugin of ExaSlang uses the Eclipse extension framework. Unlike all other elements of the ExaSlang IDE it actually depends on Java 8 not on Java 7, as it uses with this version newly introduced features. If Java 8 is not present or configured correctly, the plugin will print an error message in the Eclipse log file at start up and will not load at all. The functionality of the other components of the IDE are not compromised by this. Currently, the plugin adds the ability to call the ExaSlang generator directly from inside the Eclipse application and to see its output in the Eclipse console.

Access to generator call wrapping function is offered in three different ways through the UI. The additional toolbar button, as well as a the new "ExaSlang" menu can be seen in figure 3.15 on page 15. Additionally the keyboard shortcut `ctrl-alt-E` is provided for the same functionality.

All of these interface elements call the `generateHandler` function that starts the build process after a confirmation dialog. The extensions for this plugin consist of three linearly dependent extensions, the UI elements specified with the `org.eclipse.ui.menus` extension, the key binding, through `org.eclipse.ui.bindings`. They generate the interface elements and link to the actual command, provided by `org.eclipse.ui.commands`. The command then is assigned to the handler from `org.eclipse.ui.handlers` that links to the handler class that is called every time one of the UI elements is activated. figure 6.4 on page 53 shows all of those elements.

As the generator part of the UI does not have to be directly extended, it is not explained

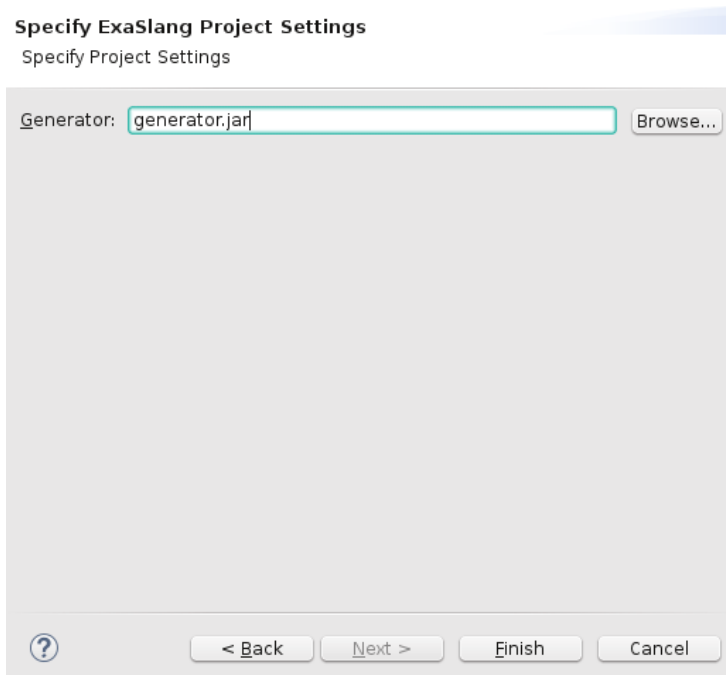


Figure 6.6: ExaSlang Project Wizard page two

here in full detail. It basically creates a process that calls the Java interpreter with a command line generated from the `generator.config` file that is present after creating an ExaSlang project and can be seen in figure 6.7. It also redirects the output of the generator process to a custom console view, exemplarily shown in figure 6.10.

```

generator.config
1 //Parameters for Generator Call
2 //Do not remove any of the Parameters
3 //You can change the Paths and Filenames as needed
4
5 generator:  "/home/stud/simiwels/thesis_michael/Generator/compiler.jar"
6 settings:  "/config/settings.settings"
7 knowledge:  "/config/knowledge.knowledge"
8 |

```

Figure 6.7: Generator.config after project generation

Further details on the generator call implementation are offered together with the Java code in the file `ExaSlangLevel4ProjectBuilder.java` inside the `ExaSlangUIElements` project. There are comments on every mayor step of the process and it is rather straight forward. In any case the plugin can be used as base for more UI functions for ExaSlang by adding additional extensions elements and specifying more commands and corresponding handlers. Please refer to the Eclipse documentation⁴ for Information on what

⁴<http://help.eclipse.org/luna/index.jsp>

the capabilities of the Eclipse UI are and how they are implemented. On page 60 figure 6.10 shows an example on how the output to the console looks after the ExaSlang generation process.

6.2.3 Plugin Deployment and Update Site

Feature and Update Site Projects

All of the ExaSlang plugins are bundled in a feature project. It consists only of a simple `.xml` file that lists all included plugins as well as information about the author, copyright agreements and so on. The deployment of the ExaSlang IDE can be done via the Eclipse internal „Install New Software...“ functionality. Whenever a new version of the IDE is deployed, the deployer simply needs to increase the version information of all changed plugins in their respective `plugin.xml` file and the version of the feature in the `site.xml`. The according wizard pages containing the version information of can be seen in figure 6.8. After that, the new feature version must be added to the `site.xml` file in the update site project. After building all plugins to update their compiled Java code to the latest version, a „build all“ on the update site project configuration page must be performed to add the new files to the update site. Both the newly added feature and the „build all“ button are shown in figure 6.9. Note that if no new version number is given, for a changed plugin, the builder assumes that the plugin has not been changed and will not overwrite the already existing version.

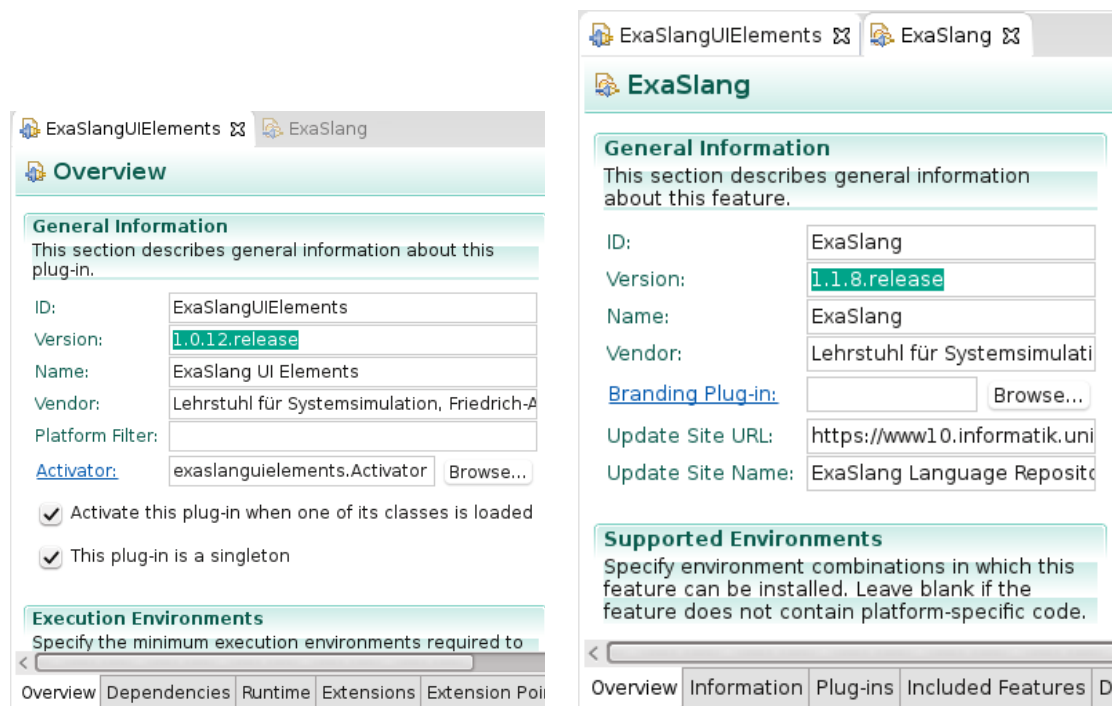


Figure 6.8: Version identifier of the ExaSlangUIElements project.xml and the ExaSlang feature.xml

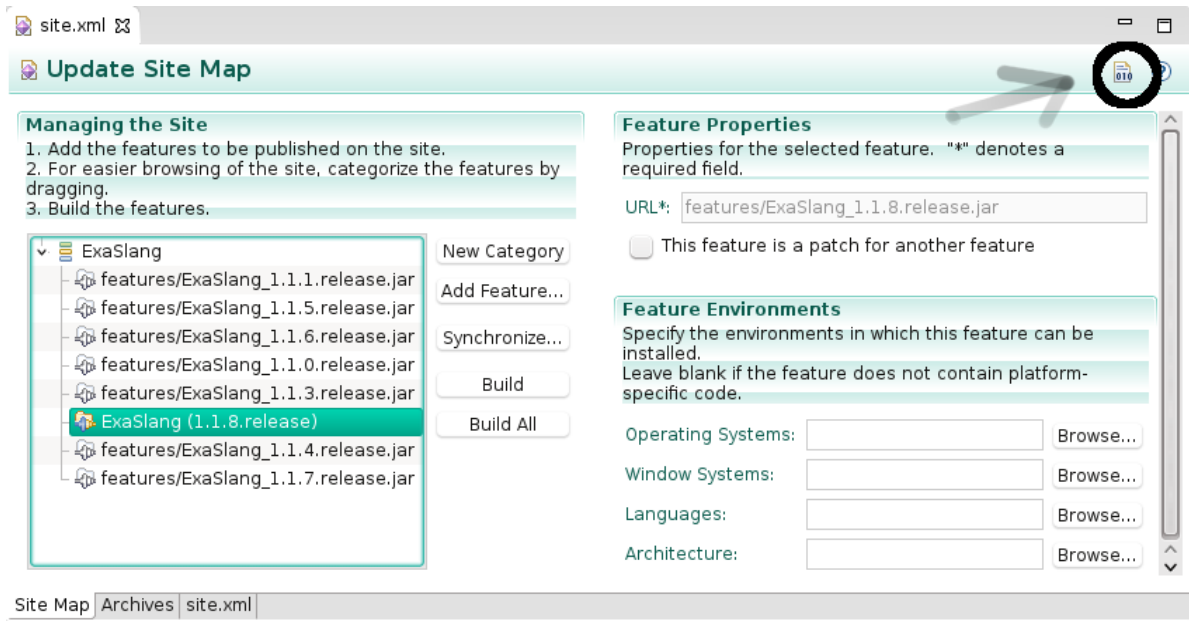


Figure 6.9: New feature version added to Update Site project, to update it the marked button is pressed

All files inside the update site project that are not marked with a leading "." can then be deployed on a server as they are. There they can be accessed by the end users to update their ExaSlang version. In figure 6.10 the web page generated by the Update Site project can be seen.

Besides automatically archiving and packaging the feature, the update site also creates a web page that links to all feature versions present on the site and allows to download them manually, if preferred over the automatic Eclipse update process, by the user. The update site project has been created with the Eclipse update site wizard and besides adding the ExaSlang feature versions has not been altered by the author. It is not expected that changes to the update site project will be necessary in the future as well.

Installation

Most parts of the ExaSlang feature package should run on Eclipse 3, but as it was created with Eclipse 4.1 and 4.2 only the latest is guaranteed to work properly when set up correctly. Note that at least Java 7, and for full functionality Java 8, is required. To install the ExaSlang IDE the "Install New Software..." feature of Eclipse can be used. It guides the user smoothly through the whole update process and makes sure compatibility issues are prevented. A detailed installation instruction can be found in appendix A.

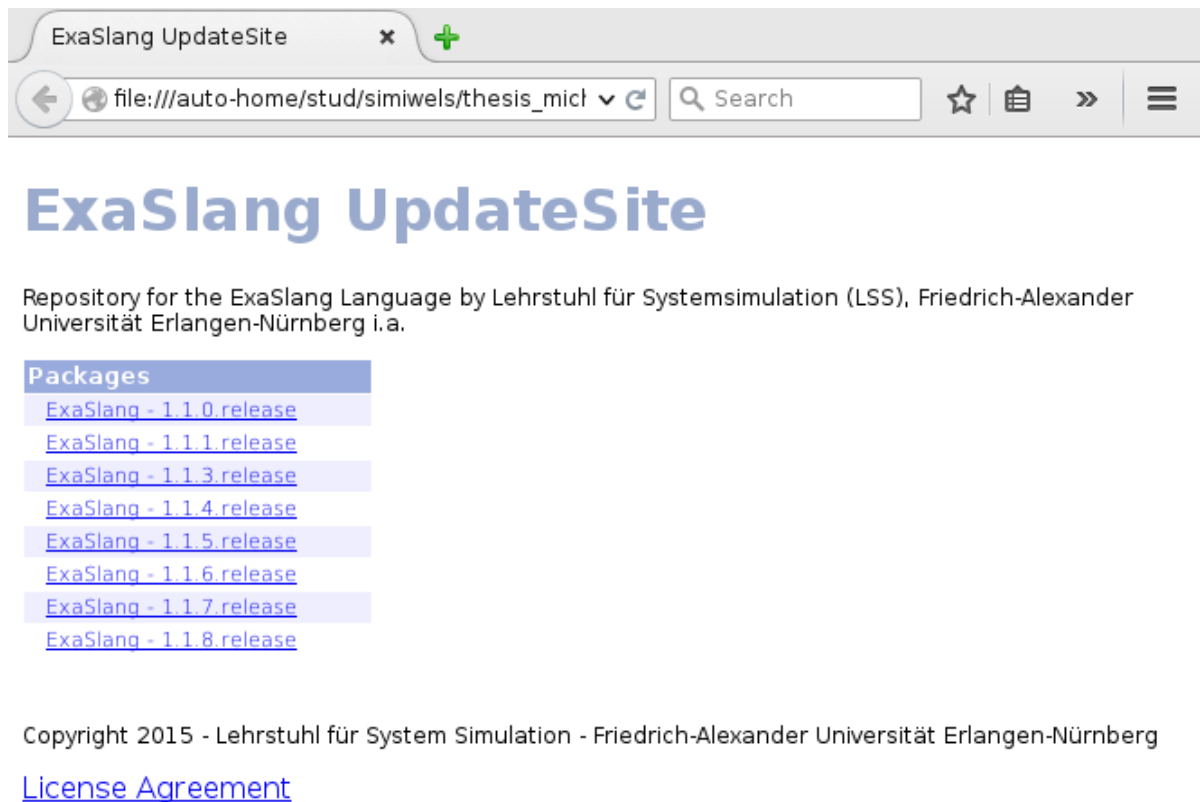


Figure 6.10: Update Site of the ExaSlang IDE

6.3 Other Features

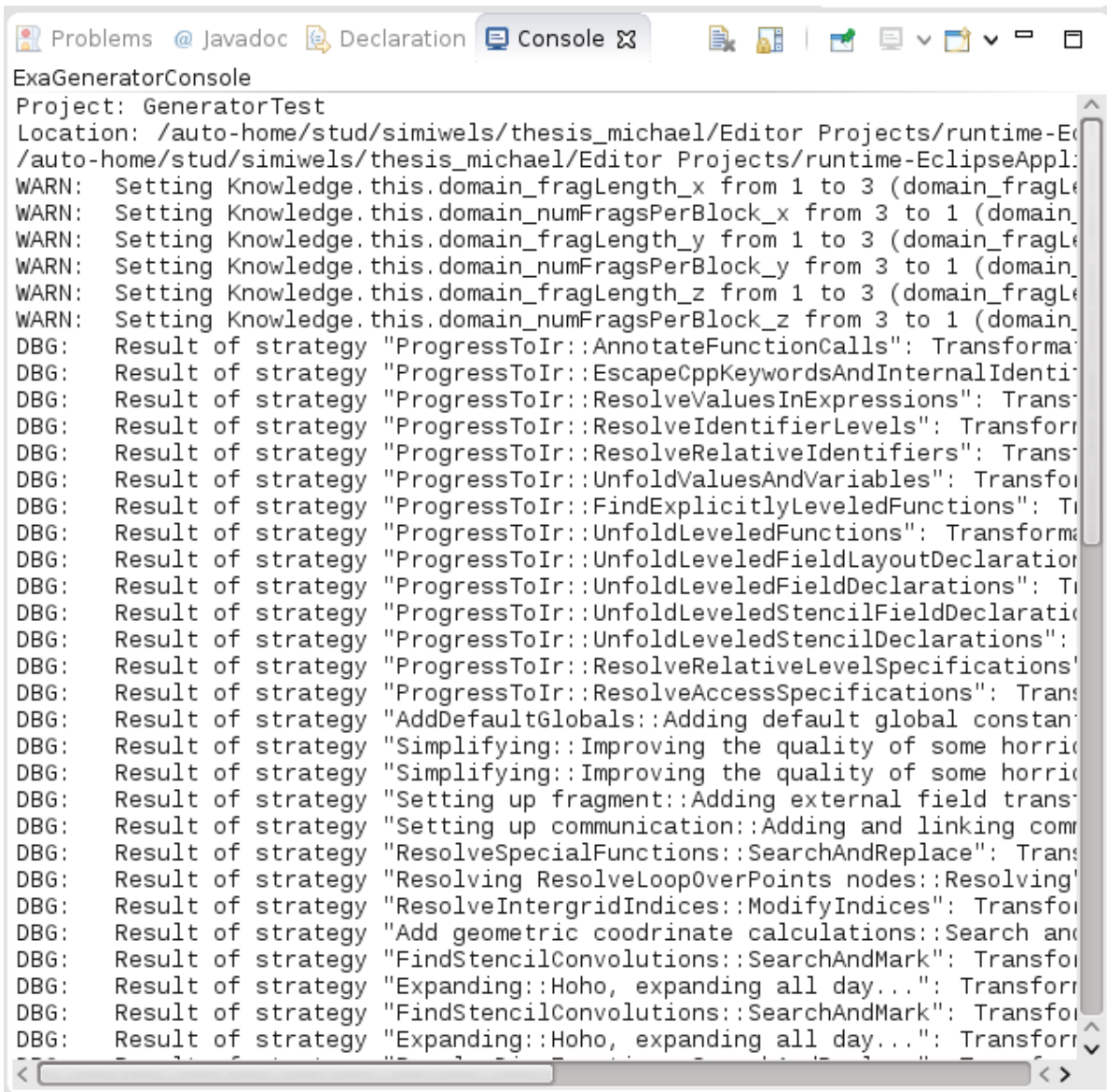
Besides the Spoofox implementation of the existing ExaSlang languages and the extension of the Eclipse framework, there are prepared project husks for the remaining, already planned, ExaSlang languages. Also included is a slightly adjusted version of the Spoofox runtime library version to avoid some key binding issues that occur during development.

6.3.1 Language Project Husks

In preparation for the future implementation of the remaining ExaSlang Languages husk projects have been prepared. Each of them is a default Spoofox project plus base files used in all ExaSlang language projects. Additionally, general bug fix and adjustment edits to the pregenerated files have been made. This allows the direct implementation of the four languages without the need of setting up and configuring the surrounding framework and ensures the consistency of all of the resulting projects with the established conventions.

6.3.2 Spoofox Runtime Adjustments

The ExaSlang extension package is delivered together with the Spoofox runtime package that is a slightly edited version of the 1.3.0.20150224-164813-master nightly build. In detail the following two changes have been made. The key binding for "go to Definition" has been remapped from `ctrl-F3` to `shift-F3` to avoid conflicts with some default key bindings occurring in some Eclipse versions. Also, a duplicated entry for the "alt-T" binding has been removed. Both of these changes are not required to run the ExaSlang package and, therefore, any version of the Spoofox runtime version 1.3 or later can be used. This is, however, not recommended due to possible compatibility issues.



The screenshot shows the Eclipse IDE's Console window. The title bar includes tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output is from 'ExaGeneratorConsole' and shows the following:

```
Project: GeneratorTest
Location: /auto-home/stud/simiwels/thesis_michael/Editor Projects/runtime-EclipseApp
Warn: Setting Knowledge.this.domain_fragLength_x from 1 to 3 (domain_fragLe
Warn: Setting Knowledge.this.domain_numFragPerBlock_x from 3 to 1 (domain_
Warn: Setting Knowledge.this.domain_fragLength_y from 1 to 3 (domain_fragLe
Warn: Setting Knowledge.this.domain_numFragPerBlock_y from 3 to 1 (domain_
Warn: Setting Knowledge.this.domain_fragLength_z from 1 to 3 (domain_fragLe
Warn: Setting Knowledge.this.domain_numFragPerBlock_z from 3 to 1 (domain_
DBG: Result of strategy "ProgressToIr::AnnotateFunctionCalls": Transforma
DBG: Result of strategy "ProgressToIr::EscapeCppKeywordsAndInternalIdentifi
DBG: Result of strategy "ProgressToIr::ResolveValuesInExpressions": Transf
DBG: Result of strategy "ProgressToIr::ResolveIdentifierLevels": Transform
DBG: Result of strategy "ProgressToIr::ResolveRelativeIdentifiers": Transf
DBG: Result of strategy "ProgressToIr::UnfoldValuesAndVariables": Transfo
DBG: Result of strategy "ProgressToIr::FindExplicitlyLeveledFunctions": Ti
DBG: Result of strategy "ProgressToIr::UnfoldLeveledFunctions": Transforma
DBG: Result of strategy "ProgressToIr::UnfoldLeveledFieldLayoutDeclaratio
DBG: Result of strategy "ProgressToIr::UnfoldLeveledFieldDeclarations": Ti
DBG: Result of strategy "ProgressToIr::UnfoldLeveledStencilFieldDeclarati
DBG: Result of strategy "ProgressToIr::UnfoldLeveledStencilDeclarations":
DBG: Result of strategy "ProgressToIr::ResolveRelativeLevelSpecifications"
DBG: Result of strategy "ProgressToIr::ResolveAccessSpecifications": Trans
DBG: Result of strategy "AddDefaultGlobals::Adding default global constant
DBG: Result of strategy "Simplifying::Improving the quality of some horrid
DBG: Result of strategy "Simplifying::Improving the quality of some horrid
DBG: Result of strategy "Setting up fragment::Adding external field trans
DBG: Result of strategy "Setting up communication::Adding and linking comm
DBG: Result of strategy "ResolveSpecialFunctions::SearchAndReplace": Trans
DBG: Result of strategy "Resolving ResolveLoopOverPoints nodes::Resolving"
DBG: Result of strategy "ResolveIntergridIndices::ModifyIndices": Transfo
DBG: Result of strategy "Add geometric coordiante calculations::Search and
DBG: Result of strategy "FindStencilConvolutions::SearchAndMark": Transfo
DBG: Result of strategy "Expanding::Hoho, expanding all day...": Transform
DBG: Result of strategy "FindStencilConvolutions::SearchAndMark": Transfo
DBG: Result of strategy "Expanding::Hoho, expanding all day...": Transform
```

Figure 6.11: Generation Console Output Example

7 Overall Results

Conclusively it can be said, that the project was successful, and almost all of its goals were achieved. Although the estimated time for the evaluation was exceeded, the implementation timeframe has been fulfilled. While most of the IDE features that have been planned were realized, these time restrains where the main reason the final goal, developing an automated process for converting existing DSLs to Spoofox Projects with a working editor, was not reached.

7.1 Implementation Time to Endresult Timesavings Is it worth the effort?

Overall, the implementation took about 200 man hours, i.e. 10 working weeks at 5 days at 4 hours each, which equals roughly 1.5 months of full-time work by a single person. This includes early bug fixes and feature changes made between version 1.0.0.release and 1.1.8.release. A quick survey at the LSS Chair of FAU, whose results can be seen in appendix C, resulted in a span of 35 % to 88 % time saved between the use of an IDE and a simple text editor for larger projects. Therefore, taking the lower number as base, a minimum of 572 man hours of total coding and debugging work would be required to make the effort worthwhile. This equals not even half a year of average 40 hour work week of a single person. For smaller projects there was no clear tendency whether it would take more or less time using the IDE. The survey was done under the assumption that the language as well as the tools are well known. Therefore it can be assumed that for new users the effects would be shifted towards more saved time when using the the IDE.

Even with those rough numbers it can clearly be seen, that implementing an IDE in the matter presented in this thesis is usually well worth its time and results, all in all, in significant time savings.

7.2 Recommendations for Future Projects

As already mentioned, the evaluation process was surprisingly time-consuming. Future projects that include an evaluation process of this kind should, in this light, make changes to the evaluation process. This could be done either by making stronger restrains upon the requirements, to speed it up by more early eliminations, or by narrowing the process down to only a few selected tools, so that those can be evaluated thoroughly. The implementation itself was exactly as initially expected, making the scope of the

implementation a perfect match for the given timeframe. Regarding Spoofax, this thesis should greatly help in future projects by providing many examples, explanations and background information brought together in one place. Spoofax is therefore the perfect tool for future projects of this kind, that do not want to undergo a new evaluation process. Especially with regard to the implementation of ExaSlang Level 1, 2 and 3 this should be kept in mind.

7.3 Feasibility for other Languages

As it has been shown, the results of the ExaSlang implementation are quite pleasing. The implementation, even of complex, usually non existent language constructs, like the Levels and Access parts, is doable. Nonetheless, Spoofax was at the brink of its capability performance wise, as ExaSlang is fairly complex and large compared to most DSLs. Therefore it can be concluded that the approach of transferring an existing DSL into a LWB, especially Spoofax, just for the IDE support is feasible. Long existing or more complex languages like ExaSlang Layer 4 profit greatly from the IDE, without the need to completely re-implement the complicated generators and transformers into another language. The time and amount of work needed to completely implement an own IDE from scratch would be significantly higher than using one of the many existing frameworks and building upon it. Not to mention, many of the frameworks are continuously developed further and enriched with new and better features. This is something a dedicated internally developed IDE is unlikely to do, given the time and manpower restrains of today's projects at many universities. Spoofax has proven to be a versatile and effective tool overall. Albeit it still requires time to accommodate oneself with it and its beta status makes it somewhat unreliable in some cases. Overall, the chosen approach was successful and can be recommended as a way to improve DSL projects in general.

8 Future Work

"Software is Never Finished, Only Abandoned."

—Unkown Software Engineer¹

This is for sure the best perspective concerning the outlook to future developments concerning the ExaSlang IDE. There are still features in Spoofax and Eclipse that can be added to the environment. Development of automation tools has not started yet. And, of course, there still are four ExaSlang languages that need implementation, Layer 1 to 3 and Hardware.

8.1 Possible Features for Future Versions

As can be seen by comparing the list of Spoofax features in chapter 5 to the list of features implemented for the ExaSlang Layer 4 IDE in section 6.1, there are still features that are missing. Most notably these are the type specification and analysis described in section 5.3 and the refactoring described in the corresponding paragraph in section 5.5. Both of these features are helpful and should be implemented if possible. Considering that work on the IDE has already been continued it is likely that the next major release of the ExaSlang IDE already has these functionalities. Unlikely to be implemented is semantic analysis, as this would imply implementing most of the code transformations to Stratego, which is not feasible according to the results of the evaluation done by Christian Schmitt et al. at in early 2014 [5]. Similarly, the implementation of automated test cases usually is not feasible for a DSL that is already existing, as all possible cases can be tested by a simple example file as well with much less effort. Other possible features for the IDE could be extended support for generation, by allowing to, for example, specify multiple configuration files that then automatically are run, and similar extensions for the ease of use.

8.2 Automation of DSL Transfer and Feature creation

Regarding the automation of the implementation of features discussed in chapter 6 it is pretty sure to say, that most of the editor features are too complex and individual to

¹after Leonardo da Vinci (Florence, 1452 a.c. to 1519 a.c.): "Art is never finished, only abandoned."

provide more than a basic husk that has to be improved manually. This husk can either be directly derived from the generated Spoofax files or be generated in a similar manner from an external tool. This only could be circumvented by largely expanding the Scala definition with additional information, like keyword and production colors, or using parts of the Scala parser as a guideline. Of course, this would make the implementation of a generator much more complex and time consuming. The language definition, on the other hand, could, maybe with the exception of complex constructs like the Access and Expression products, be generated on its own directly from the Scala files. It is even possible to assume that the additional extensions to the definition could be pregenerated in order to work with all editor features. This could possibly even make them unnecessary with a non-redundancy reducing approach to the generator.

8.3 Implementing the other ExaSlang Languages

As briefly mentioned in section 2.2, ExaSlang, once finished, will consist of four main and three support languages. Only the three existing ones have been implemented so far. Once the other four languages are close to completion it is planned that they will be implemented as well. While it will not require a new evaluation process it still will require time to learn how Spoofax works and of course time to implement the languages. Given the experience of the already done implementations one to three months of full work time should be expected for the main languages and a few weeks for the hardware language. This depends on the status of the automation tools and the depth and amount of features wanted to be realized.

9 Conclusion

The goals of the thesis have been reached. The evaluation process and its requirements have been proven to fulfill the specified goals, with regard to finding a LWB or similar tool that is capable of implementing an existing DSL without the DSL being fully implemented inside the LWB. As the amount of time used for the actual evaluation process was much larger than expected, it was concluded, that future approaches should be done by more than one person.

Spoofax, the winner of the evaluation, was used to implement the IDE for ExaSlang Layer 4 and the configuration languages, which was successfully done as well. Due to time constraints not tested implementation wise, the question, whether it is possible to fully automate the implementation can be answered with a "no". Nonetheless it is certain that partial automation is possible and should be used for future work to save working time. Despite this time intensive work a small survey indicates, that it is still worthwhile to create such an IDE in any case. Especially, because the overall conclusion is, that the chosen approach can be used for any DSL, no matter how complex.

Bibliography

- [1] Martin Fowler. *Domain-specific languages*. Pearson Education, 2010. Chap. 2.
- [2] Christian Lengauer et al. “ExaStencils: Advanced Stencil-Code Engineering”. English. In: *Euro-Par 2014: Parallel Processing Workshops*. Ed. by Luís Lopes et al. Vol. 8806. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 553–564. ISBN: 978-3-319-14312-5. DOI: 10.1007/978-3-319-14313-2_47. URL: http://dx.doi.org/10.1007/978-3-319-14313-2_47.
- [3] Christian Schmitt et al. “ExaSlang: A Domain-specific Language for Highly Scalable Multigrid Solvers”. In: *Proceedings of the Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*. WOLFHPC ’14. New Orleans, Louisiana: IEEE Press, 2014, pp. 42–51. ISBN: 978-1-4799-7020-9. DOI: 10.1109/WOLFHPC.2014.11. URL: <http://dx.doi.org/10.1109/WOLFHPC.2014.11>.
- [4] C. Lengauer et al. *ExaStencils.org*. Access: 2015/04/08. URL: <http://www.exastencils.org/>.
- [5] Christian Schmitt et al. “An evaluation of domain-specific language technologies for code generation”. In: *Computational Science and Its Applications (ICCSA), 2014 14th International Conference on*. IEEE. 2014, pp. 18–26.
- [6] Rex Bryan Kline and Ahmed Seffah. “Evaluation of integrated software development environments: Challenges and results from three empirical studies”. In: *International journal of human-computer studies* 63.6 (2005), pp. 607–627.
- [7] Sebastian Erdweg et al. “The State of the Art in Language Workbenches”. English. In: *Software Language Engineering*. Ed. by Martin Erwig, RichardF. Paige, and Eric Van Wyk. Vol. 8225. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 197–217. ISBN: 978-3-319-02653-4. DOI: 10.1007/978-3-319-02654-1_11. URL: http://dx.doi.org/10.1007/978-3-319-02654-1_11.
- [8] Christoph Ficek et al. *MontiCore 2.2.0 Framework zur Erstellung und Verarbeitung domänenspezifischer Sprachen*. Tech. rep. RWTH Aachen, Software Engineering, Nov. 2013. URL: <http://www.monticore.de/doc/MCDoku.pdf>.
- [9] *Information technology – Syntactic metalanguage – Extended BNF*. ISO Standard. Geneva, CH: International Organization for Standardization, 1996.

- [10] Karl Trygve Kalleberg and Eelco Visser. *Spoofax: An Extensible, Interactive Development Environment for Program Transformation with Stratego/XT*. Tech. rep. Delft University of Technology, Software Engineering Research Group, 2007.
- [11] Eleco Visser et al. *MetaBorg.org*. Access: 2015/04/08. 2000-2015. URL: <http://metaborg.org/>.
- [12] J. Heering et al. “The Syntax Definition Formalism SDF—Reference Manual—”. In: *SIGPLAN Not.* 24.11 (Nov. 1989), pp. 43–75. ISSN: 0362-1340. DOI: 10.1145/71605.71607. URL: <http://doi.acm.org/10.1145/71605.71607>.

List of Figures

2.1	Concept of ExaStencils [2]	3
2.2	The DSL Layers of ExaSlang [3]	4
3.1	Example Editor: Vim	6
3.2	Example for an IDE: The Eclipse environment	7
3.3	Screenshots comparing the same part of code once without and once with active Syntax Highlighting	8
3.4	The same code, once with and once without proper indentation	9
3.5	Editor snippet showing Highlighting of matching brackets in Eclipse	9
3.6	Erroneously spelled keywords highlighted by Syntactical Error Detection	10
3.7	Error detection found not expected literals	10
3.8	Template menu offering the choice of two different forms of the same construct after typing the first few letters	10
3.9	Error when trying to redefine already existing function	11
3.10	Editor reporting a non existent reference after a typo at the variable call	11
3.11	Choice of multiple variables in the template view	12
3.12	Dynamically generated tool tip	13
3.13	Outline view of an Exa4 file	13
3.14	A newly generated ExaSlang project in Eclipse	14
3.15	New Toolbar Icon and Options Menu for ExaSlang in Eclipse	15
4.1	XText: Excerpt of the long list of generated classes for the test implementation of Exa4	24
4.2	Spoofax: Excerpt from the documentation website of NaBL	26
4.3	Spoofax toolbar menu used by the Spoofax environment itself	27
5.1	Spoofax properties view showing the AST properties of the main node of a short <code>.exa4</code> file	37
5.2	Exemplary <code>.exa4</code> file converted by Spoofax generated pretty printer	40
6.1	A Global structure with several variable defines and its resulting outliner	44
6.2	Knowledge Completion Menu - three steps to a complete entry	46
6.3	Expanded and folded function in a <code>.exa4</code> file	52
6.4	Extension wizard page of the ExaSlang UI project plugin.xml	53
6.5	ExaSlang Project Wizard page one	56
6.6	ExaSlang Project Wizard page two	57
6.7	Generator.config after project generation	57

6.8	Version identifier of the ExaSlangUIElements project.xml and the ExaSlang feature.xml	58
6.9	New feature version added to Update Site project, to update it the marked button is pressed	59
6.10	Update Site of the ExaSlang IDE	60
6.11	Generation Console Output Example	61

List of Tables

4.1	List of all LWBs and Tools examined	16
4.2	List of all Requirements sorted by importance	17
4.3	Tools and LWBs, background shaded after evaluation step they reached. Darker means earlier elimination.	20
4.4	Evaluation: Final Candidate Feature Overview	21

List of Snippets

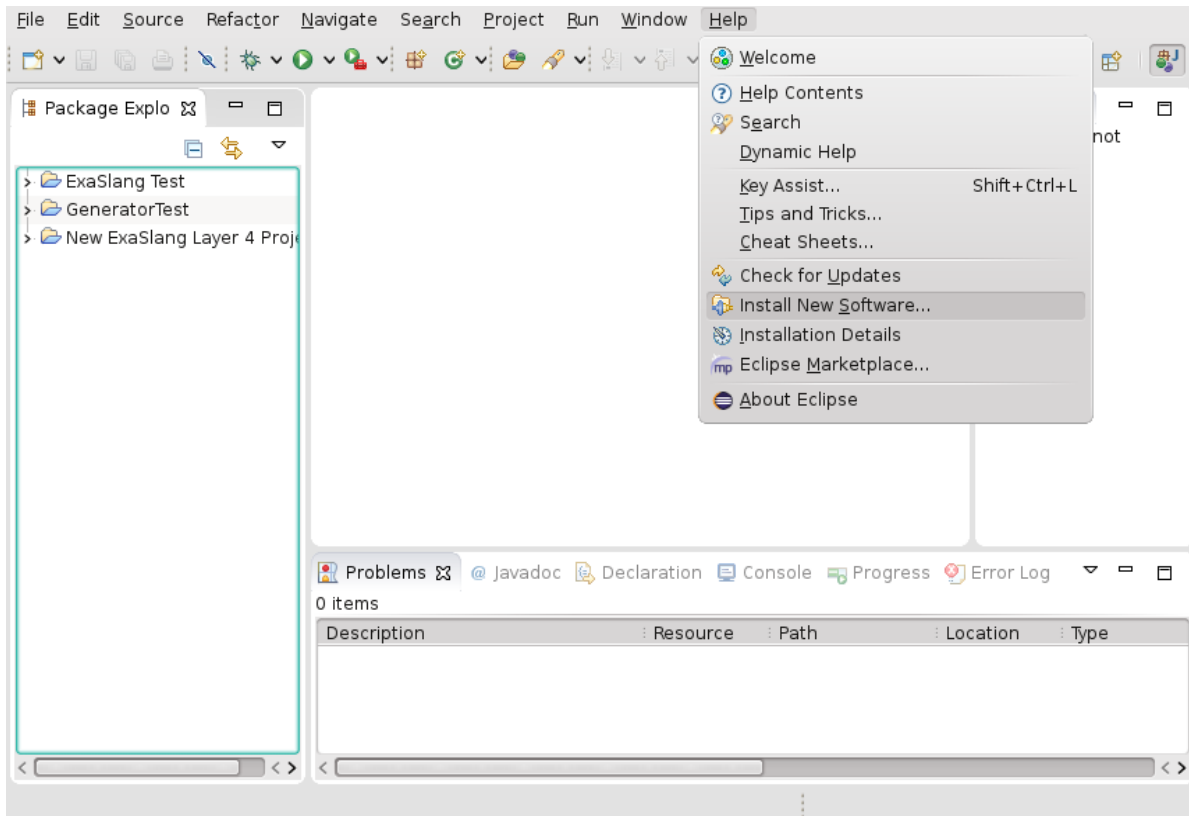
4.1	Monticore language definition	20
4.2	Editor function definition in Monticore	21
4.3	Example of XText's language definition syntax	22
4.4	XText: Example of badly done comments and documentation	23
4.5	Spoofax language definition in SDF2	25
4.6	Spoofax language definition in SDF3	25
5.1	Spoofax: Example of a module being included in another module	30
5.2	SDF3: Exemplary language definition	32
5.3	SDF3: Start symbols for example language	33
5.4	SDF3: Identifier definition and longest-match restrictions	33
5.5	SDF3: Priorities example from http://metaborg.org/sdf3/	33
5.6	TS: Type specification example from metaborg.org/ts/	34
5.7	SPT: Language testing example from metaborg.org/spt/	35
5.8	ESV: Pregenerated exa4-Views.esv file	36
5.9	ESV: Refractoring example from metaborg.org/spoofax/tour/	37
5.10	ESV: Menus example	37
5.11	ESV: Snippet from a default syntax file	38
5.12	ESV: Snippet from the unchanged main file of Exa4	38
5.13	Example of a Stratego function as it appears in Spoofax	39
6.1	Original Exa4 SDF3 definition	41
6.2	Extended Exa4 SDF3 definition	41
6.3	ESV Colorer: Example for color assignment to productions	43
6.4	ESV Colorer: Definition of Colors	43
6.5	ESV Colorer: Redefining color names	43
6.6	Stratego Outline: Generation of an outline string	44
6.7	Stratego Outliner: Extraction of information from deeper nodes in the AST	45
6.8	ESV Completions: Templates for keywords	47
6.9	ESV Completions: Examples for completion templates with multiple levels	48
6.10	NaBL: List of all namespaces in Exa4	49
6.11	NaBL: Scoping example	50
6.12	NaBL: Variable name definitions	50
6.13	NaBL: Variable name resolving	51
6.14	NaBL: Multiple choice name resolving	51
6.15	NaBL: Definition of internal variables	51

6.16	ESV Folding: Basic folding examples	52
6.17	Project Templates: Definition of DSL folder and containing exa4	54

Acronyms

AST	Abstract Syntax Tree
CoD	Chair of Hardware Software Co Design
DSL	Domain Specific Language
EBNF	Extended-Backus-Naur-Form
ESV	Editor Services Language
FAU	Friedrich-Alexander University Erlangen-Nürnberg
FOSS License	Free and Open-Source Software License
GLPL	GNU Lesser Public License
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDENT	Identifier
LSS	Chair of System Simulation
LWB	Language Workbench
NaBL	Name Binding Language
SDF	Syntax Definition Formalism
SDF3	Syntax Definition Formalism v.3
SPT	Spoofax Testing Language
TS	Type Specification Language
UI	User Interface

A ExaSlang IDE Installation Instructions



In the Help menu of Eclipse, select "Install New Software...".

Available Software

Check the items that you wish to install.



Work with:

Find more software by working with the ["Available Software Sites"](#) preferences.

type filter text

Name	Version
✓ Packages	
✓ ExaSlang	1.1.8.release

< >

1 item selected

Details

This Eclipse Feature provides IDE (Editor) support for the ExaSlang

☒ Show only the latest versions of available software ☒ Hide items that are already installed

☒ Group items by category What is [already installed?](#)

☐ Show only software applicable to target environment

☐ Contact all update sites during install to find required software

Enter the URL or path to the update site into the search field, select the ExaSlang feature group and make sure that "Show only latest version of available software" as well as "Group items by category" are checked.

Review Licenses

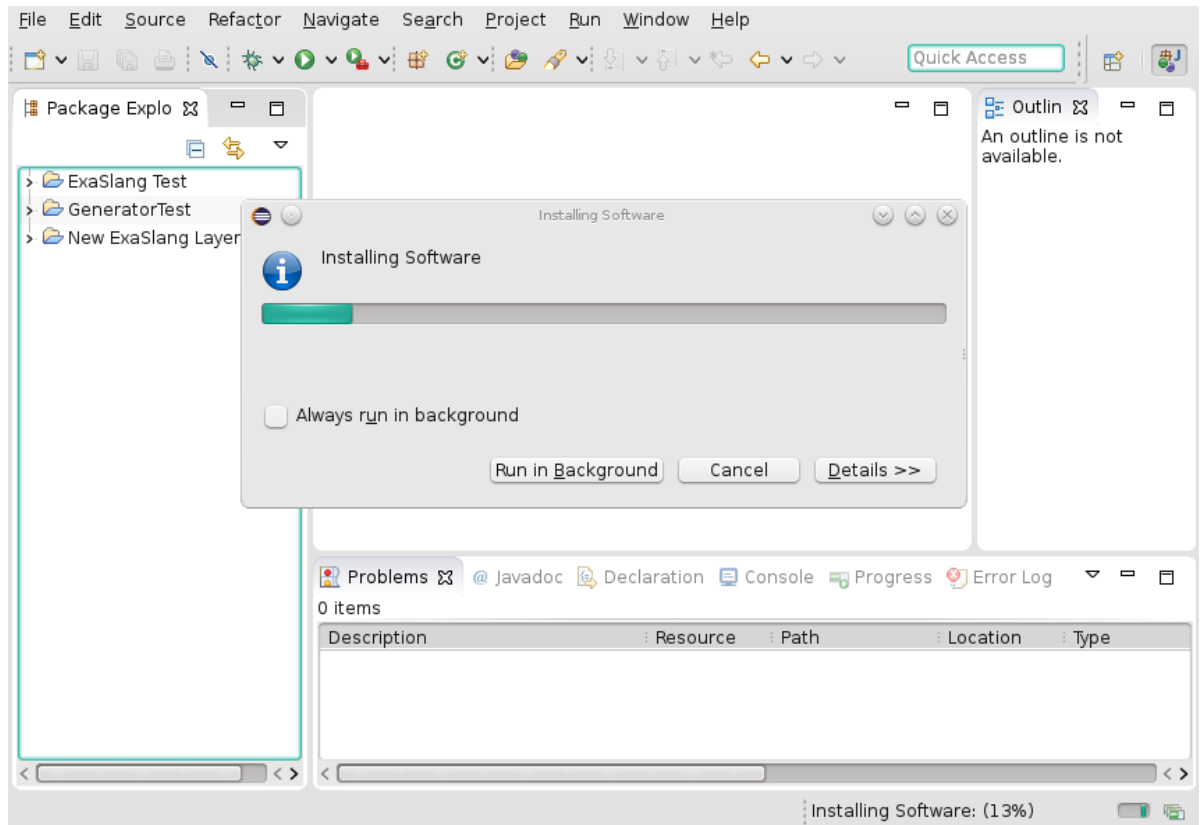
Licenses must be reviewed before the software can be installed. This includes licenses for software required to complete the install.



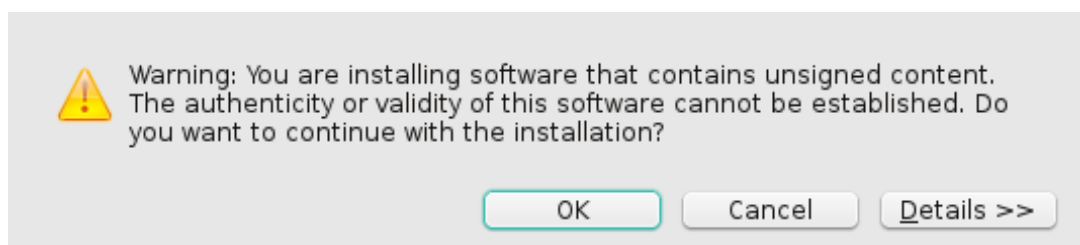
Licenses:	License text:
<ul style="list-style-type: none">> ExaSlang License Information> Spoofax License Information	<p>ExaSlang License Information</p> <p>This software is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR PARTICULAR PURPOSE. See the full license texts for more details.</p> <p>The ExaSlang Feature is licensed under the GNU Lesser General Public License (LGPL, http://www.gnu.org/licenses/lgpl-3.0.en.html).</p> <p>The Feature does NOT include the ExaStencils Generator, even if it is distributed within the same Package (Download).</p> <p>This Product uses runtime Components of Spoofax/IMP that are mostly licensed under the GNU Lesser General Public License (LGPL). Additional runtime components related to the IMP, EMF and GMF libraries are licensed under the Eclipse Public License (EPL, https://eclipse.org/org/documents/epl-v10.php).</p> <p>This product includes software developed by the Ant Contrib project (http://sourceforge.net/projects/ant-contrib).</p> <p><input checked="" type="radio"/> I accept the terms of the license agreements</p> <p><input type="radio"/> I do not accept the terms of the license agreements</p>

? < Back Next > Finish Cancel

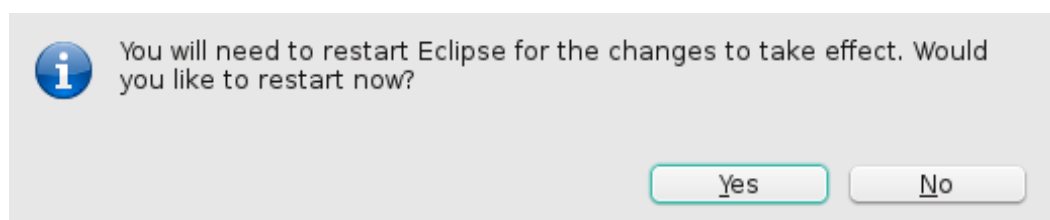
Accept the EULA for ExaSlang and Spoofax.



Wait for the installation process to finish.



When you are prompted about unsigned content click OK to continue.



Restart Eclipse and you are ready to go.

B ExaSlang License: GLPL¹

GNU LESSER GENERAL PUBLIC LICENSE

Version 3, 29 June 2007

Copyright (C) 2007 Free Software Foundation, Inc. <<http://fsf.org/>>
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

This version of the GNU Lesser General Public License incorporates
the terms and conditions of version 3 of the GNU General Public
License, supplemented by the additional permissions listed below.

0. Additional Definitions.

As used herein, "this License" refers to version 3 of the GNU Lesser
General Public License, and the "GNU GPL" refers to version 3 of the GNU
General Public License.

"The Library" refers to a covered work governed by this License,
other than an Application or a Combined Work as defined below.

An "Application" is any work that makes use of an interface provided
by the Library, but which is not otherwise based on the Library.
Defining a subclass of a class defined by the Library is deemed a mode
of using an interface provided by the Library.

A "Combined Work" is a work produced by combining or linking an
Application with the Library. The particular version of the Library
with which the Combined Work was made is also called the "Linked
Version".

The "Minimal Corresponding Source" for a Combined Work means the
Corresponding Source for the Combined Work, excluding any source code
for portions of the Combined Work that, considered in isolation, are
based on the Application, and not on the Linked Version.

The "Corresponding Application Code" for a Combined Work means the

¹<http://www.gnu.org/licenses/lgpl-3.0.en.html>

38 object code and/or source code for the Application, including any data
39 and utility programs needed for reproducing the Combined Work from the
40 Application, but excluding the System Libraries of the Combined Work.

41

42 1. Exception to Section 3 of the GNU GPL.

43

44 You may convey a covered work under sections 3 and 4 of this License
45 without being bound by section 3 of the GNU GPL.

46

47 2. Conveying Modified Versions.

48

49 If you modify a copy of the Library, and, in your modifications, a
50 facility refers to a function or data to be supplied by an Application
51 that uses the facility (other than as an argument passed when the
52 facility is invoked), then you may convey a copy of the modified
53 version:

54

55 a) under this License, provided that you make a good faith effort to
56 ensure that, in the event an Application does not supply the
57 function or data, the facility still operates, and performs
58 whatever part of its purpose remains meaningful, or

59

60 b) under the GNU GPL, with none of the additional permissions of
61 this License applicable to that copy.

62

63 3. Object Code Incorporating Material from Library Header Files.

64

65 The object code form of an Application may incorporate material from
66 a header file that is part of the Library. You may convey such object
67 code under terms of your choice, provided that, if the incorporated
68 material is not limited to numerical parameters, data structure
69 layouts and accessors, or small macros, inline functions and templates
70 (ten or fewer lines in length), you do both of the following:

71

72 a) Give prominent notice with each copy of the object code that the
73 Library is used in it and that the Library and its use are
74 covered by this License.

75

76 b) Accompany the object code with a copy of the GNU GPL and this license
77 document.

78

79 4. Combined Works.

80

81 You may convey a Combined Work under terms of your choice that,
82 taken together, effectively do not restrict modification of the
83 portions of the Library contained in the Combined Work and reverse

engineering for debugging such modifications, if you also do each of the following:

- a) Give prominent notice with each copy of the Combined Work that the Library is used in it and that the Library and its use are covered by this License.
- b) Accompany the Combined Work with a copy of the GNU GPL and this license document.
- c) For a Combined Work that displays copyright notices during execution, include the copyright notice for the Library among these notices, as well as a reference directing the user to the copies of the GNU GPL and this license document.
- d) Do one of the following:

- 0) Convey the Minimal Corresponding Source under the terms of this License, and the Corresponding Application Code in a form suitable for, and under terms that permit, the user to recombine or relink the Application with a modified version of the Linked Version to produce a modified Combined Work, in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.

- 1) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (a) uses at run time a copy of the Library already present on the user's computer system, and (b) will operate properly with a modified version of the Library that is interface-compatible with the Linked Version.

- e) Provide Installation Information, but only if you would otherwise be required to provide such information under section 6 of the GNU GPL, and only to the extent that such information is necessary to install and execute a modified version of the Combined Work produced by recombining or relinking the Application with a modified version of the Linked Version. (If you use option 4d0, the Installation Information must accompany the Minimal Corresponding Source and Corresponding Application Code. If you use option 4d1, you must provide the Installation Information in the manner specified by section 6 of the GNU GPL for conveying Corresponding Source.)

5. Combined Libraries.

130 You may place library facilities that are a work based on the
131 Library side by side in a single library together with other library
132 facilities that are not Applications and are not covered by this
133 License, and convey such a combined library under terms of your
134 choice, if you do both of the following:

135

136 a) Accompany the combined library with a copy of the same work based
137 on the Library, uncombined with any other library facilities,
138 conveyed under the terms of this License.

139

140 b) Give prominent notice with the combined library that part of it
141 is a work based on the Library, and explaining where to find the
142 accompanying uncombined form of the same work.

143

144 6. Revised Versions of the GNU Lesser General Public License.

145

146 The Free Software Foundation may publish revised and/or new versions
147 of the GNU Lesser General Public License from time to time. Such new
148 versions will be similar in spirit to the present version, but may
149 differ in detail to address new problems or concerns.

150

151 Each version is given a distinguishing version number. If the
152 Library as you received it specifies that a certain numbered version
153 of the GNU Lesser General Public License "or any later version"
154 applies to it, you have the option of following the terms and
155 conditions either of that published version or of any later version
156 published by the Free Software Foundation. If the Library as you
157 received it does not specify a version number of the GNU Lesser
158 General Public License, you may choose any version of the GNU Lesser
159 General Public License ever published by the Free Software Foundation.

160

161 If the Library as you received it specifies that a proxy can decide
162 whether future versions of the GNU Lesser General Public License shall
163 apply, that proxy's public statement of acceptance of any version is
164 permanent authorization for you to choose that version for the
165 Library.

C Survey: Time Saved by IDEs

The survey was conducted by asking ten coworkers of LSS how large they estimate the time saved, or lost, when using an IDE like Eclipse compared to a simple text editor like Notepad++ on differently sized projects.

As can be seen in table A the answers vary greatly, especially for simple projects. However, a clear tendency towards a time gain with an IDE on large projects can be seen. These observations are reinforced by the basic statistical analysis shown in table B.

Simple Project	Large Project
60 %	99.9 %
40 %	90 %
25 %	80 %
0 %	70 %
-10 %	66 %
-15 %	60 %
-30 %	60 %
-50 %	45 %
-50 %	30 %

Table A: Relative time saved or lost in different project sizes when using an IDE compared to a simple text editor. Numbers are sorted by value and not by interviewee

	Simple Project	Large Project
Arith. Mean	-3.33 %	61.59 %
Median	-10 %	66 %
σ	39.76 %	26.15 %
σ range	-43.9 % to 36.43 %	35.44 % to 87.74 %

Table B: Baseline Statistical Analysis of the Survey Results