MASTER'S THESIS | LUND UNIVERSITY 2016

Integrating Xtext and JavaRAG: Using an attribute grammar library in a language workbench

Emin Gigovic, Philip Malmros

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-22

Integrating Xtext and JavaRAG: Using an attribute grammar library in a language workbench

Emin Gigovic
dat11egi@student.lu.se

Philip Malmros
dat11pma@student.lu.se

June 13, 2016

Master's thesis work carried out at
the Department of Computer Science, Lund University.

Supervisors: Jesper Öqvist jesper.oqvist@cs.lth.se

Examiner: Görel Hedin, gorel.hedin@cs.lth.se

Abstract

Having a specialized editor or IDE has become commonplace for many programming languages. Smaller languages, especially domain-specific ones that normally have very narrow usage areas, often lack such convenience features due to their naturally small user base. Tools for implementing editor support for these languages are called language workbenches. Unfortunately these often lack features for more advanced semantic analysis, as they must be able to handle a wide range of language specifications. Reference attribute grammars (RAGs) can be used to formulate powerful semantic analysis and might, if integrated with a language workbench, help alleviate this problem.

JavaRAG is a library that can be used to add RAGs to Java based projects, which means it should be possible to integrate with the language workbench Xtext, which is built on Java. This thesis has evaluated this integration of JavaRAG into Xtext to see how beneficial the addition of RAGs would be when constructing an editor. To do this we implemented three editors, two relatively equivalent ones for a simple language where one only used Xtext while the other also made use of JavaRAG. The last editor covered a subset of a more complex language, where more advanced parts of JavaRAG could be used.

Finally we concluded that JavaRAG could be integrated into Xtext without issue, and that it offered beneficial functionality for more complex error-checking problems where Xtext's own features were sometimes lacking.

Keywords: Xtext, JavaRAG, JastAdd, domain-specific language, editor

Acknowledgements

We would like to thank our supervisor Jesper Öqvist for his extensive help during this thesis, even helping us with this report just hours before he was going on a trip to the US. We also have to thank Görel Hedin for helping us finalize the report during the last few weeks.

Contributions

The work for this thesis was divided quite equally between us. Every design choice was thoroughly discussed together before any implementation. Philip has been more involved in the development of the static analysis and JavaRAG integration, while Emin focused more on the IDE features of Xtext and defining the grammars for the languages we used in our case studies. Other work like planning, data collection, research and the report has been evenly distributed between the two of us.

Contents

1 Introduction **9**

2 Background **11**
2.1 Abstract Syntax Tree 11
2.2 Xtext 12
2.2.1 Xtend 14
2.3 JavaRAG 14
2.4 Case study 16
2.4.1 The simple language: JastAdd ASTs 16
2.4.2 The complex language: JastAdd aspect modules 17

3 Implementation **19**
3.1 Pure Xtext editor for simple language 20
3.1.1 The Xtext grammar 20
3.1.2 Static analysis 20
3.1.3 Syntax highlighting 22
3.1.4 Content Assist 24
3.2 Combined Xtext/JavaRAG editor for simple language 25
3.2.1 JavaRAG utilization 25
3.2.2 Outline view 26
3.2.3 Formatting 27
3.3 Combined Xtext/JavaRAG editor for complex language 27
3.3.1 The Xtext grammar 27
3.3.2 Static analysis 28
3.3.3 Integration with Java 29

4 Evaluation **31**
4.1 Xtext 31
4.1.1 Integrating with JavaRAG 32
4.2 Editors 37
4.2.1 Analysis 37
4.3 Lines of code 38

5 Related Work **41**

6 Discussion and conclusion **43**
6.1 Future work 44

Bibliography **45**

Chapter 1
Introduction

A *domain-specific language* (DSL) is a programming language that in contrast to general purpose languages like Java and C++ has a specific usage area where it excels, at the cost of not being able to do much else [1]. As implementing a DSL essentially means implementing a new language, albeit relatively small, there is a need for tools that make this process as easy as possible. These tools are so called *language workbenches*, with a few examples being Spoofax [2], EMFText [3] and Xtext [4]. However, even with these tools, it is difficult to formulate some standard static semantic that goes beyond the most fundamental things like duplicate name declarations. *Reference attribute grammar* (RAG) [5] systems, which are good at specifying static semantics, could therefore be a valuable addition to these tools.

The focus of this master thesis has been to evaluate the possibility to add RAG-support for Xtext. In order to do this we used *JavaRAG* [6], a library that enables the use of RAG concepts in Java-based projects. This was one of the main reasons we chose Xtext as our language workbench, as it is built using Java. We started with constructing two editors for a simple domain-specific language. One of the editors was to use nothing but Xtext functionality, while the other would make use of JavaRAG to see if there were any problems combining it with Xtext. When it had been established that JavaRAG indeed could work with Xtext we implemented the foundation of an editor for a more complex language in order to test some other parts of JavaRAG, and see how well it performed when put into a more complicated Xtext project.

The simple language that we implemented editor support for first is used by the compilation system *JastAdd* [7] to construct so called .ast files that specify an abstract grammar. The complex language is also used by JastAdd, but it handles so called .jrag files instead, that are used to specify attribute rules.

When we had the editors in place we did an evaluation to see if the inclusion of JavaRAG actually had been worthwhile. Essentially everything that can be implemented through JavaRAG is already supported in Xtext in some form, or can be implemented with only Xtext functionality. It is mainly a question of how extensive the code has to be in order to achieve the same results. So we had to evaluate how and when we should use JavaRAG instead of only native Xtext to solve a problem. Some of the things we had to consider when doing this were things such as performance, readability vs code size and extensibility.

The result of our case studies showed that JavaRAG could be integrated well with Xtext. While by no means replacing all the standard Xtext functionality, JavaRAG could effectively be used in tandem with it to bolster some areas where Xtext would normally struggle on its own. Specifically this concerns the semantic analysis of a language, and then usually the more advanced parts of this analysis.

This report is divided into the following chapters: chapter 1 gives a general overview of what the project is about, and in chapter 2 we discuss relevant background on abstract syntax trees, Xtext, JavaRAG, and our case study languages. Chapter 3 describes how we implemented the various coding-related parts of the project. In chapter 4 we present how well JavaRAG integrated with Xtext, as well as the benefits of JavaRAG compared to only using Xtext. Chapter 5 discusses alternative solutions and other tools. Finally chapter 6 summarizes the report and discusses future work.

Chapter 2
Background

There exists several language workbenches that can be used to develop DSLs with different sets of available features depending on what framework is used. Examples include Spoofax [2], EMFText [3] and Xtext [4]. Spoofax is based on the high-level SDF [8] grammar formalism, containg features like syntax highlighitng and code folding. Error handling and content assist can also be implemented using an extern language called Stratego [9]. Another language workbench, EMFText, is based on the the Eclipse Modeling Framework (EMF) [10]. Currently there exists a collection of 100 concrete text syntaxes available on the EMFText website, which can be used for further development or inspiration. Xtext which is also based on EMF with the support for the majority of the features stated above. It should be considered that the EMFText editor does not seem to be updated as frequently as Spoofax or Xtext. We choose to use Xtext as our language workbench over the other ones mentioned above due to it being most documented, maintained, and its tight integration with Java.

2.1 Abstract Syntax Tree

An *abstract syntax tree* (AST) is the abstract syntactic representation of a program, where each node in the tree corresponds to a language construct in the source code. ASTs are often used by compilers and language workbenches to construct an internal representation of a program that only contains the essential information of how the program's source code is structured. Figure 2.1 and 2.2 show an example of how source code could be represented by an AST.

```
1        if (a > b) {
2            a = a + 2;
3        } else {
4            a = a + 1;
5        }
```

Figure 2.1: Example code with AST representation in figure 2.2.

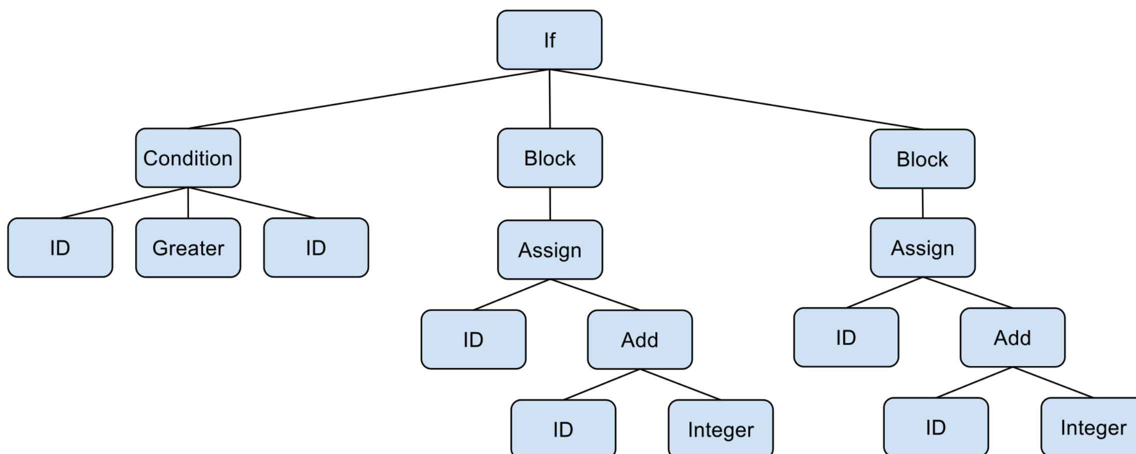


Figure 2.2: AST representation of the code in figure 2.1.

Figure 2.3 shows part of a simple grammar (in EBNF [11]) that covers the code in Figure 2.1. It should be noted that this grammar is very simplified and hard-coded to work as a small example. It is written specifically to not need the more general productions an actual grammar would have to represent something like an if-statement.

```
1    If -> "if" "(" Condition ")" Block ["else" Block]
2    Condition -> Expression Operation Expression
3    Block -> "{" Assign* "}"
4    Assign -> ID "=" Add
5    Add -> Expression "+" Expression
```

Figure 2.3: Grammar in EBNF form covering the AST in figure 2.2 with corresponding code in figure 2.1.

2.2 Xtext

Xtext is an open source framework based on the Eclipse IDE, used for developing programming language editors. Functionality such as static analysis, syntax coloring, code completion and an outline view can be implemented either by only using native Xtext tools or by combining the functionality from the existing Xtext tools, or with external libraries.

The Xtext grammar is defined by the user to describe the concrete syntax of a language and how it is mapped to the AST, in Xtext referred to as the semantic model. An advantage with the Xtext framework is that it automatically handles the creation of the AST and creates corresponding Java classes to store the AST with defined rules for each class. The generation of the AST is executed by EMF, which provides code generation support for building tools and applications based on structured data models. The grammar consists of rules for each language construct, and for each rule an EMF interface and class is generated by Xtext. The generated fields for each feature defined by the rules are combined with getters and setters. Figure 2.4 shows the grammar production for the if-statement from the first line of code in figure 2.3. In figure 2.5, the corresponding generated interface is shown, presenting the generated methods for the `If` production.

```
1 If:
2     'if' '(' condition = Condition ')'
3     block = Block
4     ('else' elseBlock = Block)?
5 ;
```

Figure 2.4: The Xtext grammar production for the if-statement described in figures 2.1, 2.2 and 2.3.

```
1 // Note that all language construction classes inherits from
     EObject, which is the root type of all the modeled objects
2 public interface If extends EObject
3   Condition getCondition();
4   Block getBlock();
5   Block getElseBlock();
6   void setCondition(Condition value);
7   void setBlock(Block value);
8   void setElseBlock(Block value);
9 {
```

Figure 2.5: The generated interface for the `If` production declared in figure 2.4.

After defining the grammar or later modifying it, the Modeling Workflow Engine (MWE2) must be run to generate an ANTLR parser and EMF classes. *ANTLR* [12] is a LL(*) parser generator used widely and by Xtext to build languages, tools and frameworks. *MWE2* is a generator used by Xtext to generate classes and corresponding methods, and derive an ANTLR specification to create the AST. The generated code is placed in the source folder *src-gen*, in which nothing should be modified, because everything in the folder will be overwritten by the next generation.

2.2.1 Xtend

By default, several of the files in Xtext are written in *Xtend* [13], a statically typed language extending Java, with a focus on more concise syntax. Xtend also supports functionality such as type inference, extension methods, lambda expressions and operator overloading. In figure 2.6 we can see the difference in creating an `ArrayList` in Java versus Xtend. The main difference is the use of `var` in Xtend and defining the type of the `ArrayList` is optional since Xtend has type inference.

```
//Create an ArrayList in Java
ArrayList<Integer> myList = new ArrayList<Integer>();

//Create an ArrayList in Xtend
var ArrayList<Integer> myList = new Arraylist<Integer>()

//or with type inference in Xtend
var myList = new ArrayList<Integer>()
```

Figure 2.6: Some differences for variable declarations in Java and Xtend.

2.3 JavaRAG

JavaRAG [6] is a Java library based on RAGs supporting computations of static properties in an AST, and can be attached to an AST if the nodes in the tree are based on Java. The most common use of RAGs is for developing modular extensible compilers and programming analysis tools.

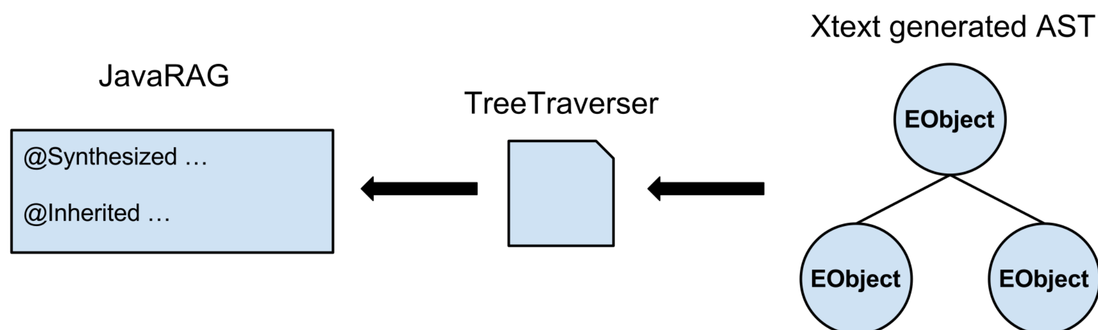


Figure 2.7: Illustrates the connection between an AST and JavaRAG, where the information from the AST is passed along to JavaRAG with the help of a TreeTraverser.

Figure 2.7 gives a general overview of how JavaRAG operates. The AST, in our case consisting of EObjects generated from Xtext, that we want to add RAG functionality to is given to a so called *TreeTraverser*. The TreeTraverser retrieves information from the AST and passes it to JavaRAG, which then calculates its attributes from the AST information and stores it in its own data structures.

The method `checkConditionIsBool` in figure 2.8 illustrates how JavaRAG can be used by Xtext's error checking.

```
1 def checkConditionIsBool(If stmt){
2    // The line below retrieves JavaRAG information
3    val evaluator = getEvaluator(stmt)
4    val isBool = evaluator.evaluate("conditionIsBool", stmt)
5    // Give an error if "isBool" is false...
6 }
```

Figure 2.8: A simple check using a JavaRAG attribute to check if the condition in an if-statement is valid (if it is a boolean value).

The same kind of attributes are used in JastAdd and JavaRAG, attributes are defined by equations and can be synthesized or inherited. A synthesized attribute is annotated as `@Synthesized` and an inherited attribute as `@Inherited`, making it clear for the evaluator how to evaluate an attribute. `@Cached` attributes perform caching to avoid unnecessary recalculations. The `@Circular` attribute may depend on itself and is evaluated through a fixed-point iteration. JavaRAG and JastAdd use the same algorithm [14] to check circular attributes, with a minor modification in JavaRAG to increase the evaluation efficiency. In figure 2.9 we can see how attributes are declared in JavaRAG, with an illustration of its corresponding AST in figure 2.10.

```
1  public class NameAnalysis extends NameAnalysis.Interface>
      extends Module<T> {
2      public interface Attributes {
3          @Inherited ClassDeclaration parent(Child self);
4          @Inherited Declaration lookup(Child self, String name);
5          @Synthesized String name(ClassDeclaration self);
6          @Synthesized String name(Child self);
7      }
8
9      // ...
10 }
```

Figure 2.9: Example code showing how attributes are declared in JavaRAG. Corresponding AST representation is shown in figure 2.10.

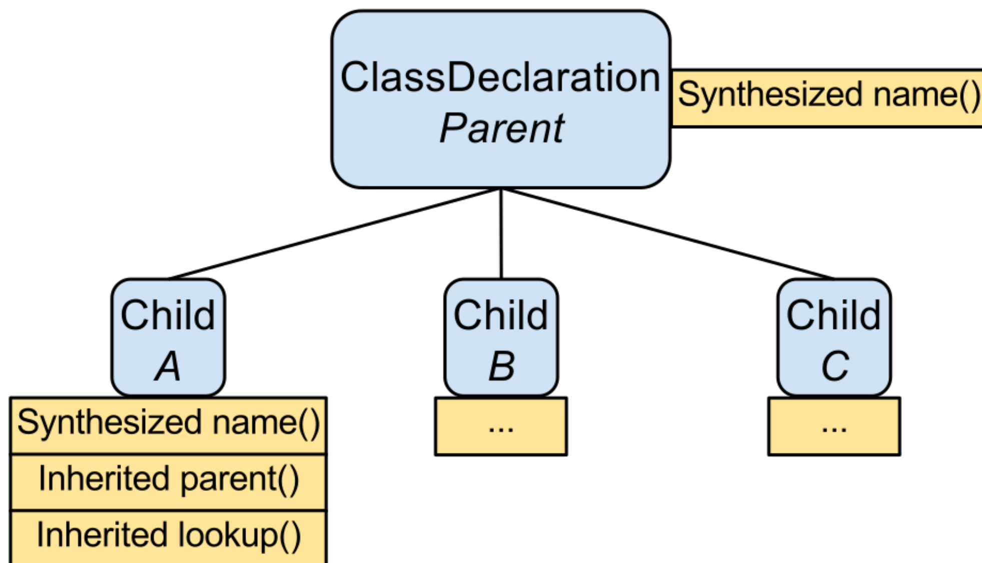


Figure 2.10: AST representation of the code in figure 2.9.

2.4 Case study

To evaluate how well JavaRAG could be integrated with Xtext, we implemented editor support for two languages as our case study. Both languages are used by JastAdd [7], a Java based meta-compilation system that supports RAGs. The first language is used for handling .ast files while the other one is used for .jrag files.

2.4.1 The simple language: JastAdd ASTs

The .ast language is relatively small, which is the main reason we chose to implement it as the first step of the thesis. It would give us a good opportunity to test the feasibility of using JavaRAG instead of pure Xtext on something smaller, instead of starting to work on the main attribute grammar part of JastAdd immediately. The usage of this language is to specify an abstract grammar by declaring attributes in AST classes. In figure 2.11 we can see an example of an abstract grammar, and figure 2.12 shows a more detailed example of what a class declaration consists of.

```
1   abstract A;
2   B : A;
3   C ::= B [E];
4   D : C ::= B;
5   E ::= <myNumber:Integer> myList:C*;
6   F ::= /G/;
```

Figure 2.11: Example of AST code.

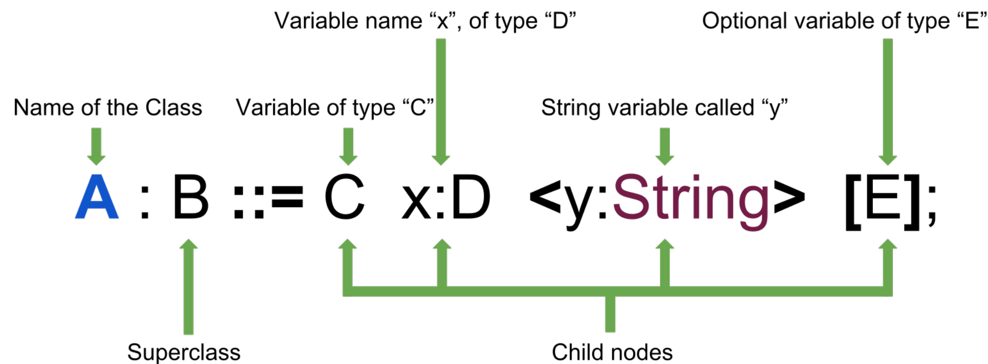


Figure 2.12: A Class declaration in the simple language.

2.4.2 The complex language: JastAdd aspect modules

An object-oriented representation of the AST combined with the use of attribute grammars is the base of JastAdd. Attributes can be defined as synthesized or inherited, depending on if the information should be propagated upwards or downwards in the AST. The equation of a synthesized attribute is defined in the same node as the attribute, while inherited attributes are defined in a child node but the corresponding equation is in an ancestor. The user can reorganize the code into modules for reuse and composition. JastAdd supports many different attribute declarations, some of which are shown in figure 2.13.

```
1  aspect ABC {
2      syn String A.x() = "myString";
3      syn nta Integer D.w() = 3;
4      syn Integer A.y();
5
6      eq A.y() = 1 + 2;
7
8      inh C B.z();
9
10     eq A.getB().z() {
11         return getC();
12     }
13
14     coll LinkedList<B> A.c();
15
16     String myString = "Hello";
17 }
```

Figure 2.13: Example of an aspect module.

Chapter 3

Implementation

In our thesis we developed three interactive editors: Two versions implementing a simple language (JastAdd AST language) where one only used Xtext functionality, and the other making use of JavaRAG. The last editor covers a subset of a more complex language (JastAdd aspect modules). Figure 3.1 shows a simple illustration of how the language workbench Xtext works. The first step is to chose a programming language that the editor is to be built for, and then construct a grammar that covers it. With the grammar completed, IDE features can be added until the editor has all the functionality that is required. JavaRAG can then be added to certain parts of the IDE features to take advantage of RAGs. This chapter will cover some of the main areas of each editor, and where applicable some comparisons will be made between them.

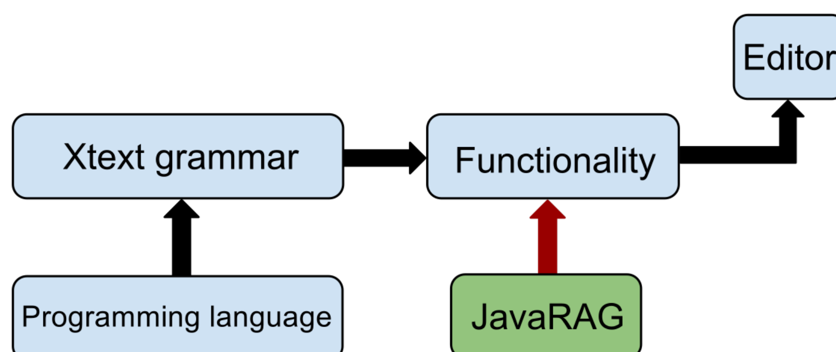


Figure 3.1: A simplified structural overview of how Xtext and JavaRAG interact.

3.1 Pure Xtext editor for simple language

To get started with Xtext, we implemented an editor for the AST language used by JastAdd. This would later be used as the baseline for what our second editor version should support. We implemented full AST language support, including the following features:

- Static analysis
- Content assist
- Syntax highlighting
- Quick-fix
- Automated tests

Something that should be kept in mind is that we didn't need to make use of all the available parts of Xtext when building our editors. For instance Xtext lets you specify a full code generator for your language, but in our case this wasn't needed, as we can simply run the code through JastAdd instead.

3.1.1 The Xtext grammar

The grammar used by Xtext to cover the simple language is very straightforward, and only consists of 9 rules spread over 28 source lines of code. When JavaRAG was used in the simple language this was increased to 12 rules and 39 source lines of code, but this was mostly to introduce some abstraction for the rules. Generally the rules are only one line long, and many of them are variations of the *Child* rule as seen in figure 3.2. The reason for this is simply that the language to cover is small, and only really needs to be able to declare classes and their potential child nodes.

```
1 Child:
2     Component | ListComponent | OptionalComponent  |
3     TokenComponent | NTA
4 ;
```

Figure 3.2: Rule for the *Child* nodes in the Xtext grammar.

3.1.2 Static analysis

In Xtext, static analysis is implemented through a so called validation file. Every time information in the editor is changed, methods in this file that have the annotation "`@Check`" and a parameter taking a class declared in the grammar, will be called. It should be noted that if a `@Check` has a parameter of the type "Abc", the `@Check` will be called for every separate instance of "Abc" that was parsed from the source code.

Having several instances calling time-consuming `@Checks` every time something changes in the editor might however put a heavy strain on performance. An alternative can then be to add a parameter to the `@Check`, informing the editor to only run it on certain events. The available parameters are *FAST*, *NORMAL* or *EXPENSIVE*, where *FAST* is the standard option and *NORMAL* only performs `@Checks` when the file is saved. In our implementation we did not make use of these parameters, as most of our `@Checks` are not very demanding, and not getting any feedback on errors apart from when a file is saved restricts the usefulness of a `@Check` drastically.

```
1    @Check
2    def checkAbcForAnError(Abc abc) {
3        // Perform some evaluation of an aspect of abc...
4        if (error was found) {
5            error(...)
6        }
7    }
```

Figure 3.3: Example structure for a `@Check`.

Figure 3.3 shows an example of how one of our `@Checks` is structured. The parameter has information specified from the grammar, and in the body of the `@Check`-method some evaluation is made to find errors or warnings. If an error or warning is found, it is reported to the user through corresponding methods, *error(...)* or *warning(...)*. Generally we followed the norm in software-testing to only test for one thing per test.

The list below gives an overview of what kind of `@Checks` we have implemented for this editor, and in figure 3.4 there are some code examples that violates a few of these `@Checks`.

- Classes and child nodes have correctly defined names
- Classes do not depend on themselves, i.e. circular inheritance
- Child nodes have correctly defined types, and do not extend their parent class

Something we had to keep in mind when implementing the static analysis was that the validation file is written in the Xtend language which has a few quirks that we had to work around. The most notable example is how Xtext makes use of `null` when assigning default values to certain objects, and how Xtend handles these `null` values in some situations. If a value in an object from the grammar is not initialized, like in the case of an optional value, it gets assigned `null`, leading to frequent null-checks in the validation class. As a special case `String` variables get assigned an actual `String` with the value “null” if uninitialized, so checks for those values have to use an `equals` method instead.

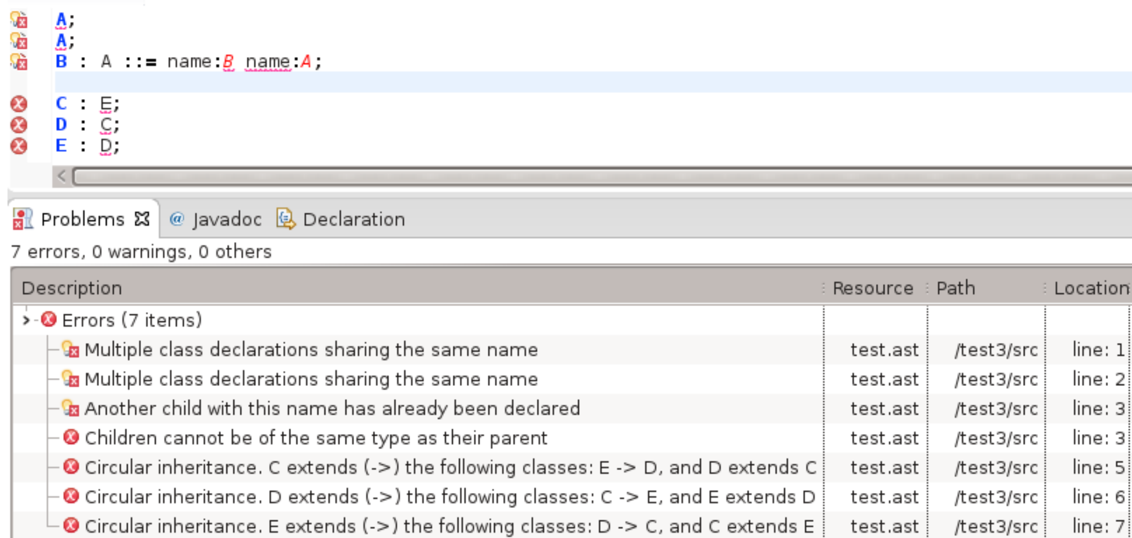


Figure 3.4: Examples that violate the validation rules for the simple language using pure Xtext.

3.1.3 Syntax highlighting

The practical benefits of syntax highlighting is the immediate feedback regarding the syntactic correctness, achieved by customizing code in different visual styles, leading to increased readability overall. Important keywords in the language are easier to distinguish and recognize, due to being highlighted in a customized color.

Our implementation of the syntax highlighting functionality is separated into two parts, semantic and lexical highlighting [15]. Highlighting customization is achieved by providing an implementation of the interface `ISemanticHighlightingCalculator`, where the AST generated by Xtext is traversed and selected node elements are highlighted in a custom style. In figure 3.5 the implementation for highlighting an element of type `ClassDeclaration` is shown, where the AST generated by Xtext is traversed and if the node is of type `ClassDeclaration`, highlighting is performed.

```
1  public class AstHighlightingCalculator implements
       ISemanticHighlightingCalculator {
2
3  public void provideHighlightingFor(XtextResource resource, ...)
       {
4          INode root = resource.getParseResult().getRootNode();
5          for (INode node : root.getAsTreeIterable()) {
6              if (node.getSemanticElement() instanceof
                   ClassDeclaration) {
7                  // Highlight the node of type ClassDeclaration
8              } else if (...)
9                  // more cases for different highlighting
10         }
11 }
```

Figure 3.5: Shows how the AST is traversed and a selected element is highlighted.

The configuration of a style customization is defined in a separate class and performed by implementing the interface `IHighlightingConfiguration`, which defines the style of a selected node element. The defined style is then assigned an unique ID, which is called in the semantic highlighting implementation when a selected node element should be highlighted. Below in figure 3.6 an example of the syntax highlighting function is illustrated for a short code example. Note that the child nodes B and C in the class declaration E are highlighted differently because E has a superclass D which has the same child nodes B and C.

```
//Abstract Syntax Tree Editor
abstract A;
B;
C;
D ::= B C;
E : D ::= B C;
F ::= <myInteger:Integer>;
```

Figure 3.6: Illustrates syntax highlighting for the simple language.

3.1.4 Content Assist

Content assist, or auto complete, is a feature providing the user with valid code suggestions in the actual context of how to complete partially typed code. Xtext auto-generates a basic content assistant with core functionality only using suggestions from existing keywords in the grammar language. We added more suggestions by extending the class `AbstractAstProposalProvider` and defining corresponding methods for each element in the grammar. Additionally we changed for which contexts the content assist gave suggestions, for instance so that it did not always present every possible option no matter what you were writing.

Our content assist helps the user to create a new grammar element by providing the user with a skeleton of the intended declaration with standard names, where the user then can change the standard names to their liking. Keyword suggestions are shown in the right context with the exception of bracket, assignment and list symbols. Proposals are given for existing declarations if the user wants to reuse them later. The autocomplete is accessed via the keyboard shortcut *Ctrl* + *Spacebar* and an example of it in use is shown in figure 3.7.

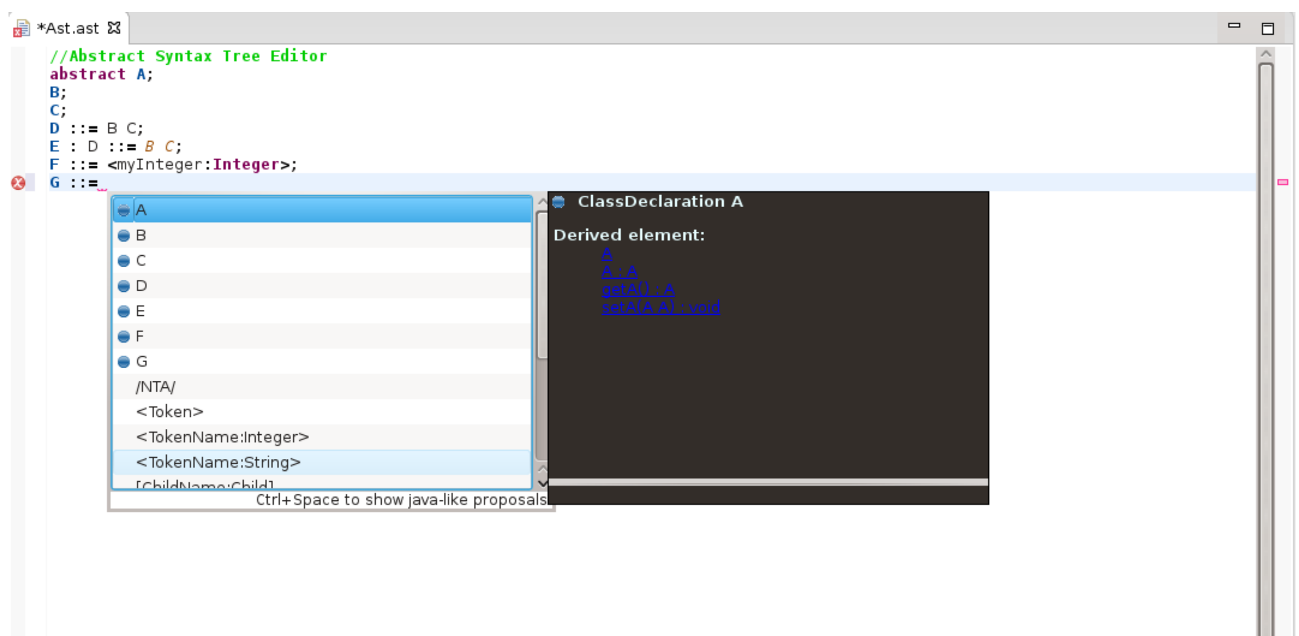


Figure 3.7: The content assist feature being used in the editor.

3.2 Combined Xtext/JavaRAG editor for simple language

To evaluate JavaRAG, we made a copy of the editor we already had, but changed the validation file to make use of JavaRAG instead. Originally we had intended to make the validation functionality completely equal between the two variants, however, compared to the first version we ended up fixing some issues and adding some functionality. IDE features that was added, compared to the pure Xtext versions was: formatting, outline view and auto generation of classes. While the editors are no longer completely equal, the evaluation of how well JavaRAG could be integrated is still valid.

3.2.1 JavaRAG utilization

As stated above, our goal was to port over the static analysis from the original editor and implement the equivalent functionality again while taking advantage of JavaRAG. We had two ways of doing this, either by simply taking the code from the `@Checks` in the first editor and put them in attributes in JavaRAG, or by generally changing the way the calculation was made by using RAG concepts.

Overall we had no significant problems with moving, or adding new, code to the JavaRAG file or calling this code from the validation class. We did however notice that while our goal was to use JavaRAG as much as possible, Xtext had already done a lot of the work for us. For instance we could have added attributes for all the important information such as names of objects or their types, and accessed these using JavaRAG. However most basic information like this was already available through simple method calls in Xtext, as any decent grammar will have parsed this information into the objects from the code it is given through the editor. This led us to opt for using the already available Xtext methods a lot of the time when possible, since there was no real point investing time into writing JavaRAG code that would always be inferior to the built in Xtext functionality. Instead we focused on trying to use JavaRAG for things that would be more difficult to do in Xtext, and that could not be solved through a simple method call. This mainly meant that we changed the implementation of the `@Checks` for correct declarations of names and circular inheritance. We also extended `@Checks` for valid names of inherited child nodes and classes or child nodes using keywords as names. Figure 3.8 shows some examples of code that violates the changed/new `@Checks`.

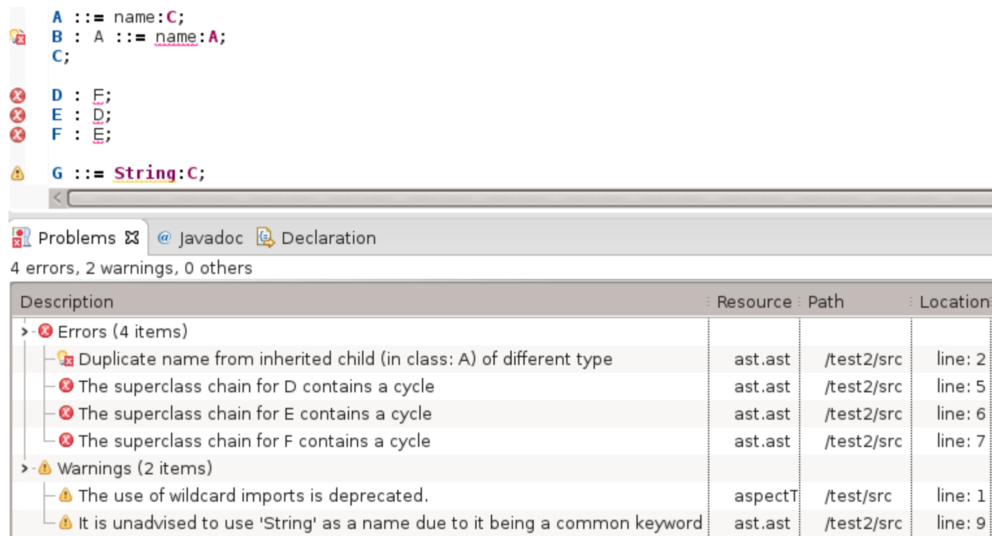


Figure 3.8: Examples that violate the validation rules for the simple language when both JavaRAG and Xtext are used.

3.2.2 Outline view

The outline view is located at the right-hand side of the editor, helping the user navigate through the model by being represented as a hierarchical view, where the elements are sorted in the order they are typed into the editor. By selecting an element in the outline view the corresponding element in the editor is highlighted. The outline view is customized by implementing an `IOutlineTreeProvider`, where each node in the tree is an instance of an `IOutlineNode`.

Since the calculations of the outline nodes are done on demand, the UI will show expandable nodes that do not actually have any child nodes if selected, so called *hidden nodes*. We solved this by overriding a specific method for each declaration that could possibly have a child node, which caused them to be revealed and represented in the outline view in a correct way. In figure 3.9 we can see the outline view feature from our editor.

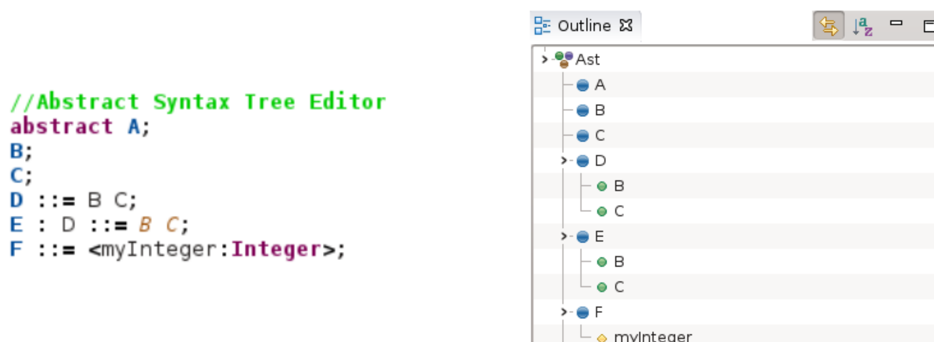


Figure 3.9: A code snippet on the left and its outline view representation on the right.

3.2.3 Formatting

The formatting feature in our editor is accessed via the keyboard shortcut *Ctrl* + *Shift* + *F*, rearranging the code in the editor for improved readability. The implementation of this feature is achieved by extending the class `AbstractDeclarativeFormatter` and specifying actions that should be applied for each declaration in the grammar language. We chose to have a newline after each declaration in the editor, unnecessary white space in a declaration is removed and the line indentation of each line is set to zero.

3.3 Combined Xtext/JavaRAG editor for complex language

Compared to the simple language versions, the complex language editor is significantly larger in scope and more complex, due to a much more extensive grammar language. In this editor we combined native Xtext and JavaRAG functionality more freely than in the simple language editor. RAG concepts were utilized more with the focus of plugging the gaps where using only native Xtext would be clunky or convoluted, while pure Xtext was used for the more straightforward `@Checks`. As mentioned in section 3.2.1 Xtext makes it easy to access information directly related to a given object. However, while it is possible to then reach most other objects by traversing the AST, writing the code for this will involve quite a few loops and conditions to access information, and of course, the time it will take to go through all these loops and conditions only increases as files become larger. By contrast, with JavaRAG we can instead use `Collections` to gather information in a node that is easily reachable by objects that need it. A `Collection` is a type of attribute that combines values defined through another type of attribute called a `Contribution`, that can be spread throughout the AST [16].

The aim for this editor was never to fully cover all of JastAdd, as this would require a significantly larger time investment. Our goal was rather to build an editor that could handle the parsing of JastAdd and contain error handling and other editor features for a smaller core part of the language. Naturally this core needed to contain enough to make the addition of JavaRAG worthwhile, and also make room for possible additions, so that a larger and eventually fully featured version could be implemented.

3.3.1 The Xtext grammar

JastAdd aspects contain RAG-specific constructs as well as plain Java code, and in section 2.4.2 in figure 2.13 we can see an example of how they are used. One of the main reasons why we used Xtext in this thesis was specifically since it lets us import functionality for Java-related things like import sections, standard types, classes and interfaces. We do this by importing Xbase [17], a statically typed expression language for Java, into the Xtext grammar. Unfortunately, like the other options we have looked at, Xbase does not support the use of pure Java code blocks. So we had to do a compromise where the code blocks that in JastAdd would normally be Java code, are replaced with blocks of Xbase code.

The Xtext grammar for the complex language consists of 22 rules spread over 112 source lines of code, compared to the simple language grammar that is described in section 3.1.1. Considering that this grammar is so much larger than the ones used in the other editors, we had to make more extensive use of the abstract functionality in Xtext's grammar language. Otherwise the code needed for the static analysis, and to an extent JavaRAG, would quickly have started to get filled with several non-trivial methods size-wise that would have to be more or less duplicated multiple times.

3.3.2 Static analysis

Due to the more complex nature of the JastAdd grammar, the code used for the static analysis has in turn increased. Not only because there are more different things that can be analysed now, but also since some types of analysis now require significantly more code to perform in relation to the simple language counterpart. An example of this would be the analysis needed to check for duplicate names in the complex language. The reason for this is that more checks are needed in order to make sure there are no duplicate names, since there are more types of named entities in the complex language.

Because of the time it would take to fully implement the static analysis for JastAdd, we decided that for this thesis we would focus on implementing validation rules covering a basic core of the aspect language. The main reason being that we don't need a full implementation of JastAdd to evaluate how helpful JavaRAG has been in the implementation of the editor. The more important rules implemented in the static analysis include:

- Return values are of correct type
- Equations must have connected attributes
- Synthesized attributes must have an equation
- Errors for duplicate names

To be a bit more specific about the use of JavaRAG, 9 of the 19 `@Checks` rely on JavaRAG to function. It should also be kept in mind that several of the pure Xtext `@Checks` are fairly simple and are only a few lines in length, and include style warnings for attribute names etc.. From the list of rules described before, it is only the `@Check` for return values that uses pure Xtext, and the other three must have access to information that is retrieved with JavaRAG. In figure 3.10 there are some examples of code that violate rules in the previously mentioned list of `@Checks`.

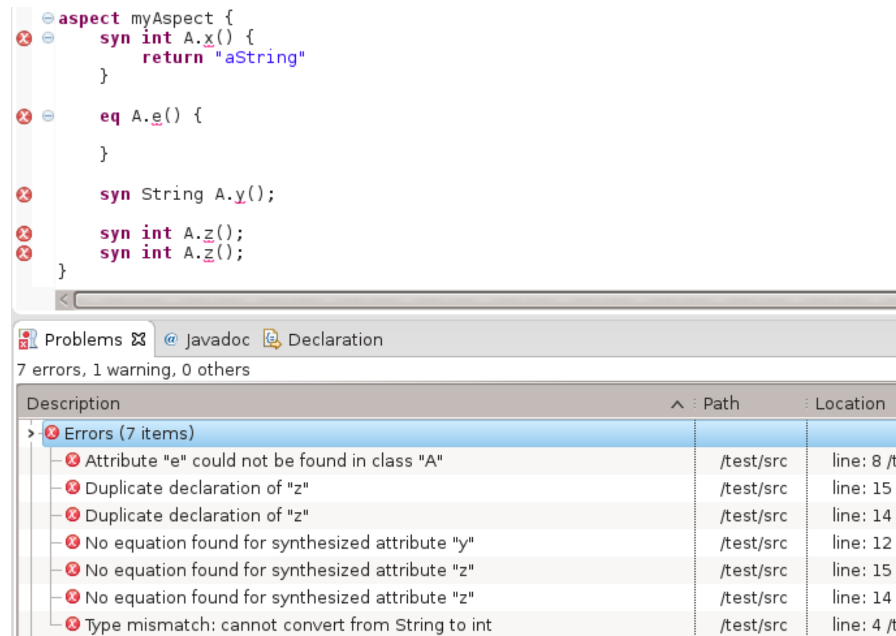


Figure 3.10: Examples that violate the validation rules for the complex language.

3.3.3 Integration with Java

In the complex language ordinary Java code can be used to construct attributes, but also independently in an aspect module. In figure 3.11 we see an example of how Java code can be used to determine the return type (String) of the `syn` attribute.

```
1 syn String A.b(){
2     var myString = "Hello"
3     return myString
4 }
```

Figure 3.11: Example code showing the use of Java and Xbase in the complex language.

To be able to use Java code in our language we used Xbase, a programming language that can be embedded and extended within DSLs written in Xtext. A JVM (Java Virtual Model) model had to be inferred, which means that we implemented the `IJvmModelInferrer` interface and defined where Java code should be allowed. For the code example in figure 3.11 we specified that the return type of the `syn` attribute is allowed to be a Java type and inside the block. Each attribute in a Java class is generated as their own classes in the *src-gen* folder, and for the code example above a class will be generated with the name `Syn_A_b.java`. The generated class consists of methods, getters, setters and fields specified in the attribute construction. As we mentioned in section 3.3.1 the consequence of using Xbase is that pure Java code cannot be used inside blocks, instead Xbase code must be used, which is similar to Java.

Chapter 4
Evaluation

In this chapter we will evaluate how well the tools we used for this thesis have performed, both from the perspective of what they have allowed us to do as well as how effectively we have used them. To start with we will describe our experiences working with Xtext, and how suited it has been for the task at hand. Then we will discuss how we combined Xtext with JavaRAG. If it was as useful as we had initially hoped, and if it could be properly taken advantage of to overall perform better than a pure Xtext implementation. After this we will give our thoughts on the state of our editor(s), how well featured they are and considerations in regards to performance and extensibility. Finally we give a general look at the size of the project's code base.

4.1 Xtext

We had no previous experience with using programs like Xtext before starting this thesis, so the workflow needed to make Xtext function in Eclipse was our first real obstacle. As there are so many different parts that can be used in Xtext, not all of them even necessary, we had to make sure that we were working in the correct areas and that everything we put into our code would function together with the other modules in Xtext. This led us to focus on a subset of the available modules that we identified as necessary to make the editor run in the first place.

Later we discovered that some problems could have been solved differently if we had used other modules that were not needed at the start to make the editor function. For instance we could have used the native `Scoping` module to make Xtext generally calculate what objects had access to other objects. If properly implemented this would have let Xtext handle most errors concerning unique names for classes etc. on its own. By comparison we solved this by having special `@Checks` in the validation class directly. Several of the modules would however remain unused as they were simply not related to, our outside,

the scope of this thesis. For example the `Generator` module would have served no real purpose, as the system for code generation is meant to be JastAdd.

As was mentioned in section 3.1.2, the Xtend language, which is used in several Xtext files, has a few traits that can take some time to get used to, in particular how it handles `null` values. Having to make use of extensive null-checking might not be an appealing solution, but it is not necessarily wrong. The problem we had was that another trait of Xtend seems to be that under some circumstances `NullPointerExceptions` will be blocked or ignored, and the program is allowed to keep running as if no exceptions were thrown. In the case of a forgotten null-check this would often result in confusion as entire methods or `@Checks` simply seemed to stop working for no apparent reason.

4.1.1 Integrating with JavaRAG

Because of the way Xtext creates several projects and then has them work together to ultimately construct the editor, we were at first worried that there might be some problems adding JavaRAG on top of all the native Xtext functionality. However, when we found where JavaRAG could be the most useful, it turned out to be surprisingly easy to integrate it into Xtext.

One aspect of the JavaRAG implementation that is suboptimal is the performance. This stems from that JavaRAG is dependant on Xtext's generated AST for its calculations, so if that AST changes, the JavaRAG information also has to be updated. The validation class is similar to a JUnit class in that on some event, for Xtext that would be an editor-change, all the methods marked with a certain annotation are run. The issue for the validation class is that it has no equivalent to the `@Before` and `@After` annotations found in JUnit. This coupled with that the JavaRAG information has to be updated on every editor-change means that every single time we want to make use of JavaRAG in a `@Check`, that `@Check` has to recalculate all the JavaRAG information it depends on. While the implementations we have worked on are small enough that this is not really a problem for us now, in the future if this editor was to cover all of JastAdd, this could turn into a critical obstacle. We have attempted a few solutions to this problem, including storing the `Evaluator` as a member variable that was only updated once every time a change occurred in the editor by using a "dirty" flag. While this might only have been an issue with our implementation, this method did not work reliably. In the end we agreed on that while we likely could implement some form of hack solution to only update the `Evaluator` once per editor-change, it would be better to keep the code correct even if it means worse performance. Finding an actually good solution to this problem is definitely something that would be on the list of future work.

Apart from the performance concerns, the addition of JavaRAG has generally felt like a worthwhile one. We found that it can be used to move code out of the validation class and into separate JavaRAG files, as shown in figure 4.1, if this actually makes the code easier to read is a matter of personal preference. The main benefit we found for JavaRAG has been the use of `Collections`. A `Collection` is an attribute that uses some form of data structure to store important information, which when needed calls all of its connected `Contribution` attributes to populate the data structure. This essentially lets us send information across the program with ease as long as the `Collection` is declared on a class that is always reachable, like the root-node of the program.

```
1  // IN VALIDATION FILE
2  def  checkForCircularInheritance(ClassDeclaration classDecl) {
3        val evaluator = getEvaluator(classDecl)
4        if (evaluator.evaluate("cycleInSuperclassChain",
            classDecl)) {
5           error(...)
6        }
7     }
8
9  // IN JAVARAG FILE
10 public boolean cycleInSuperclassChain(ClassDeclaration self) {
11       // True if self has a a cycle on its superclass chain
            (actual method is 21 lines long)
12 }
```

Figure 4.1: Example of how JavaRAG can be used to refactor out code as if a normal method call had been made, but with the benefit of RAGs.

One of the main annoyances we had implementing parts of the simple language editor using pure Xtext was that accessing data not directly linked to an object required loops and conditions for often relatively simple checks. Here the `Collections` from JavaRAG turned out to be a significant boon. Consider the following: we want to perform a check on an attribute that requires the names of all other similar attributes, like when we want to see if the attribute's name has already been used. We have found three ways to do this. The first and most brute force one being to use Xtext methods to cast a net covering every visible object and then filtering out the attribute declarations we need. This works in something smaller like the simple language editor when looking for duplicate names, but seems very taxing otherwise. The second method is to crawl through the AST with standard loops and conditions, which could look something like the code in figure 4.2.

```
1  def checkDuplicateAttributeNames(ABC abc) {
2      val model = EcoreUtil2.getRootContainer(abc) // Get access
           to root-node
3      for (aspect : model.aspectElement) {
4          for (statement : aspect.statement) {
5              if (/*statement is in same namespace as ABC*/) {
6                  if (statement.name.equals(abc.name) {
7                      // If this happens twice, a duplicate has
                           been found, can for instance be done with
                           a counter
8                  }
9              }
10         }
11     }
12 }
```

Figure 4.2: A possible way that the AST could be crawled through in order to find dublicate declarations of `abc`.

The third option is to use `Collections`. Figure 4.3 shows how we set this up, with the list of attribute names being accessed simply by calling the "`attributes`" attribute. The advantage that the third option using `Collections` has here is that it does not require any kind of information filtering which will only get slower as the amount of code to process increases. Instead there is a direct connection between the data structure to populate, and the entity possessing the information that the data structure wants. While it takes some overhead to set the `Collection` up initially, if done properly it is also easy to reuse between `@Checks` that need the same information.

```
public interface Interface {
    // Declares the Collection in the root-node (Model)
    @Collected List<Declaration> attributes(Model self);
}

// Mainly decides what kind of data structure to use
public CollectionBuilder<List<Declaration>, Declaration>
    attributes(Model self) {
   return new CollectionBuilder<List<Declaration>,
      Declaration>(new ArrayList<Declaration>());
}

// Declares a Contribution for all objects of the type "ABC"
public void attributes(ABC self, Collector<Declaration> col) {
   Model node = (Model) EcoreUtil2.getRootContainer(self);
   col.add(node, self);
}

// Other Contributions for types also considered to be
    attributes
```

Figure 4.3: All the necessary code to set up a `Collection` that contains a list of all the declared attributes in the program.

JavaRAG also made it possible to implement a *lookup pattern* [5] to find multiple declarations of classes etc.. This pattern basically means that an object it is invoked on "looks up" in the AST to see if there are any objects that match a certain criteria, a name for example. In the case of a match the matched object is returned, and if this isn't the same object the lookup was invoked on, a duplicate has been found. See figure 4.4 for a simple code example of the "lookup pattern".

```java
public void methodA() {
    int a;
    if (...) {
        int a;
    }
}

public void methodB() {
    int a;
}
```

Figure 4.4: Illustrates the lookup pattern with Java code. If the lookup is invoked on the `a` inside the `if` it will "look up" in the AST and ask `methodA` if there are any declarations with the name `a` there, which there is. If the lookup had been done on the `a` in `methodB` instead the lookup would not find either `a` from `methodA`, since `methodB` will not find anything in itself, and Java methods cannot look in each other. See figure 4.5 for a more graphical representation.

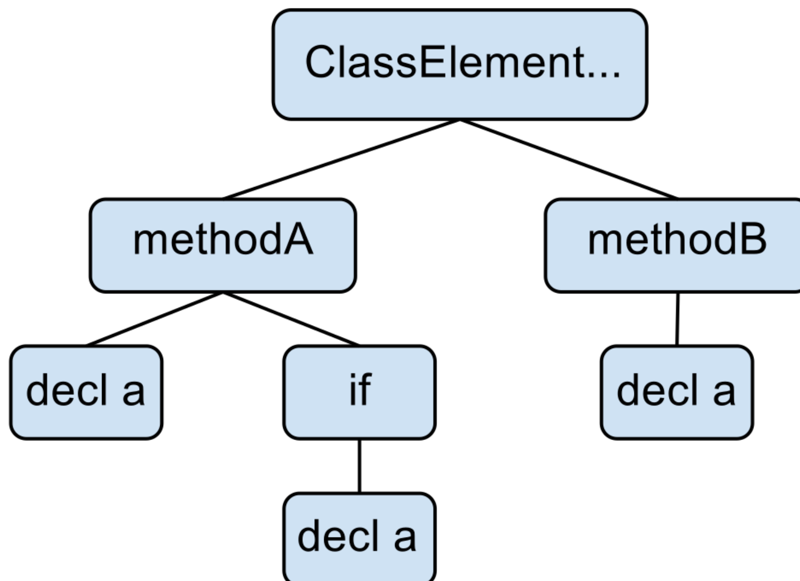


Figure 4.5: A simplified AST for the code in figure 4.4

4.2 Editors

As mentioned in the start of chapter 3 we implemented two separate editors for the simple language and one editor for the complex language. Our objective with this thesis was to evaluate how well JavaRAG integrated with Xtext, therefore we utilized JavaRAG as much as possible where we found it suitable compared to native Xtext functionality. The main difference between the two editor versions of the simple language is in the static analysis part, where we took advantage of JavaRAG functionality to implement our validation rules.

4.2.1 Analysis

The syntax highlighting, content assist, formatting and outline view modification features in our editor(s) were implemented with the functionality offered by Xtext. The reason being that these features did not seem to benefit from functionality offered by JavaRAG, and so we felt that native Xtext was more appropriate to use here. This is because the methods in these features do not need access to more information than is declared in the grammar for any object, and this is easily retrieved using Xtext methods, making JavaRAG superfluous.

While Xtext offers good support for accessing information of a specific node object, the node can have a parent node and zero or multiple child nodes connected to it. Accessing information that is connected to a node located further down or up in the hierarchy can be demanding depending on how far away the information is located. For example consider a node which has a list of child nodes, and each of these children in turn has children, and so on. If we want to access an object located a bit further down in the hierarchy, we have to first retrieve the child list from the start node, iterate through that list and find our next connection node. We will then have to repeat this procedure until we find the node we were actually searching for. If now each list is of size n, then at worst we have to iterate through n elements in a list and this procedure is done at worst m times, where m is the number of steps we iterate up or down in the hierarchy. The complexity or the performance won't be an issues if m is a small number, indicating that we access an object located near the start node. This is not quite optimal, and therefore when more demanding and complex property access is needed, we felt is was better to utilize JavaRAG `Collections` as described in section 4.1.1.

As stated in section 3.3.3 we used Xbase to be able to use Java code in the complex language, which resulted in classes being auto-generated for each attribute construction. There exists a lot of space for optimization and customization of this functionality, what should be generated and how it can be used. We noticed a minor bug with this functionality on the current Xtext version, where we sometimes had to refresh the *src-gen* project folder to display the newly auto-generated classes, but apart from that the function worked well.

We tested how well our editor for the simple language could handle larger files by using input from ExtendJ [18], which is an extensible compiler for Java. The largest AST file was in the range of 500 lines of code and our editor handled it without errors, noticeable bugs or significant lag. The aspect editor was tested manually, and checked to see if the code was valid or not according to the syntax. Due the use of Xbase inside the expression blocks rather than Java, it was difficult to use existing aspect files for testing.

4.3 Lines of code

We made a quantitative comparison in the form of lines of code with the evaluation tool Cloc [19]. Cloc counts SLOC (Source Lines of Code), i.e., the amount of code excluding blank lines and comments of a chosen programming language. In table 4.1 we can see the amount of code for the implementation of the static analysis for both editors of the simple language. An additional file called *NameAnalysis* in this case, has to be included in the JavaRAG version, which contains the JavaRAG functionality. Comparing the lines of code for the two versions of the simple language, we can see that the one including JavaRAG is almost twice as large. The reason for this size difference is that JavaRAG needs some setup code, which is approximately 70 - 100 SLOC. Another reason is that the combined Xtext/JavaRAG version contains a few more implemented features than the pure Xtext version, which may result in the versions not being relatively comparable. The intention is to show an approximate difference in SLOC when JavaRAG is added.

	File	Code	Total
Pure Xtext	Validator	230	230
Combined Xtext/JavaRAG	Validator	254	
	NameAnalysis	201	455

Table 4.1: Source lines of code for the simple language editor.

Table 4.2 shows the lines of code for the static analysis implementation of the editor for the complex language. The *Validator* file is understandably larger compared to the simple language editor, but the Nameanalysis file is also noticeably smaller. This is mainly because the Nameanalysis code uses JavaRAG very differently for the two languages. In the simple language editor a "lookup pattern" is used, while in the complex language editor `Collections` are used more extensively, and in general `Collections` take less space than the lookup pattern. The reason why we used different approaches was that during the progression of our thesis work we learned more about how JavaRAG could be used, which resulted in that the JavaRAG functionality was better utilized in the complex language editor.

File	Code
Validator	433
NameAnalysis	142

Table 4.2: Source lines of code for the complex language editor.

An absolute value of the amount of code for all the files that we implemented for respective editor is shown in table 4.3 and in figure 4.4 the number of SLOC is shown for each feature in respective editor. This gives us a brief overview of the size for the implemented editors. Xtext generates several files when a Xtext project is created, which are not included in our measurement.

Editor	Code
Simple language with pure Xtext (1)	975
Simple language with Xtext/JavaRAG (2)	1591
Complex language with Xtext/JavaRAG (3)	1230

Table 4.3: Total source lines of code for respective editor.

Feature	1	2	3
Grammar	28	39	112
Static analysis	230	455	575
Syntax highlighting	187	204	-
Content Assist	64	64	-
Quick-Fix	66	157	-
Tests	400	524	-
Formatting	-	51	110
Outline View	-	77	121
JVM Inferrer	-	20	312

Table 4.4: Source lines of code for each feature in respective editor.

Chapter 5
Related Work

Today there exists a number of different DSL language workbenches to develop DSLs with both simple and more advanced features. Below we will give a brief overview of a set of academic efforts that have been published in this domain, and also how RAGs can be applied and used in different areas.

Bettini, Stoll, Völter, and Colameo, 2012 [20] proposed several approaches and tools for implementing type systems in Xtext. The main purpose of their evaulation was to conclude the flexibility, required effort and usability of respective approach and tool. Their comparison was based on three alternative approaches to implement type systems in Xtext, named XSemantics [21], Xtext Type System (XTS) [22] and Xbase. The conclusion of their study stated that Xbase was very useful when the DSL was tightly integrated with Java, because of the full integration support with the Java type system. Meanwhile XSemantics and XTS provided a framework and a DSL to make the type system implementation concise and more maintainable, regardless of the DSL had any connection to Java or not.

Buerger, Karol, Wende and Assmann, 2010 [23] approached a concept to integrate metamodelling languages like EMF with RAGs. Because of the lack of support for formal semantic in metamodelling they developed JastEMF. JastEMF is a tool used for specifying the semantics of an EMF metamodel by using JastAdd RAGs. Given an EMF metamodel the semantics are specified by integrating generated code from JastAdd and EMF, with the help of JastAdd RAGs. Their approach confirms that RAGs can be integrated with meta-models such as EMF. JastEMF differs from Xtext in such way that it is centralized around adding RAGs to specify semantics for a meta-model, while in Xtext RAGs are not added directly to the meta-model. The development of JastEMF is based on using generated code from JastAdd, which is a drawback in such way that if JastAdd is updated the current JastEMF version will not work properly.

Name Binding Language (NaBL) [24] is Spoofax's own DSL, used for the specification of name bindings and scope rules. Name binding is the relation between definitions and references in a language, while scoping rules are used to restrict the visibility of a definition. NaBL uses an algorithm to automatically generate IDE features such as error marking for unresolved references, constraint checks and code completion. Compared to the development of our editors and the integration of Xtext and JavaRAG, some of these features were manually implemented and others automatically included by Xtext. Error marking for unresolved references was automatically included as a feature by Xtext, while the code completion feature was implemented from scratch.

Chapter 6
Discussion and conclusion

This thesis work was based on the idea of combining the JavaRAG library with the language workbench Xtext and evaluating the benefits of this combination. This was done largely in two steps. First two relatively small editors were implemented for the .ast files used by JastAdd, where one of them took advantage of JavaRAG to solve part of the implementation, while the other did not. The second step was to make an editor for a more advanced language used by .jrag files in JastAdd that would be more able to take advantage of the features offered by JavaRAG.

During the first step we did an evaluation of the feasibility of integrating Xtext with JavaRAG. This was done so that when we started working on the editor for the complex language we could be assured that problems encountered generally would not stem from trying to make use of JavaRAG. While we had initially been worried about adding JavaRAG functionality into the Xtext project structure, this turned out to be a non-issue. Instead we were surprised at how few problems we had when making use of JavaRAG. When issues with JavaRAG did arise, it was mostly contained within the JavaRAG files, and not concerning the combination of JavaRAG and Xtext. While we were expecting that JavaRAG would mainly be used for the static analysis, we also considered other areas where it might be useful, like the content assist or quick-fix features. We did not find that JavaRAG offered any better solutions in these areas than pure Xtext however, so in the end JavaRAG was used exclusively in the static analysis.

After we had established that JavaRAG could be used by Xtext during the first part of the thesis, we started working on the editor for the complex language, where we found more clear advantages of JavaRAG. In the first two smaller editors most of the functionality we needed could be covered by Xtext without any problems, but in the third one this started to change. Xtext had problems with accessing certain data easily when performing the error-checking, but JavaRAG offered solutions with more readable and concise code assuming the reader knows how RAGs work. Up until this point the addition of JavaRAG had felt

like nice a concept, but we had struggled to find any particular aspect where it felt clearly superior to a solution offered by Xtext, but the data-access methods offered by RAGs had now finally changed this.

In the end we can with confidence conclude that JavaRAG generally functions well with Xtext, and while maybe not immediately useful for simple problems concerning error handling, it makes a good addition for more complex tasks. The main challenge when utilizing JavaRAG is finding where and how to use it to get a better solution than with standard Xtext functionality.

6.1 Future work

There are several parts of Xtext that we did not use when constructing our editors, and one of them is the code generator. While we did not want to implement this from scratch, we would want to connect it to JastAdd, for making the use of the editors more seamless than having to compile the code separately. Another feature that has been mentioned earlier in section 4.1 is `Scoping`, which if implemented would have the potential to make some of the code in the validation class simpler.

Naturally the parts that we have implemented can be further extended as well, for instance the validation for all the editors so that more advanced errors can be found. The JastAdd editor's syntax highlighting, content assist and quick fix features could also be extended to not only cover the most basic cases.

A limitation with developing the editor for the complex language, mentioned in section 3.3.1, was the lack of support for pure Java code in blocks and expressions, even though Xbase was used which is based on Java. A future improvement to our editor is to implement support for pure Java code. To do this a plugin named Jbase [25] can be used instead of Xbase. Jbase is a variant of Xbase, able to handle pure Java code in expressions and statements. The Jbase plugin is currently in a development state and only supports older Xtext versions, with the porting to newer Xtext version underway. The transition from Xbase to Jbase in our implementation would not require much work due to Jbase's similarities with Xbase. The required work would be managing the Jbase installation, such as a correct classpath setup as well as in the grammar language file changing the top line from:

```
1 grammar se.lth.cs.jastaddxtext.aspect.Aspect with Xbase.Xbase
```

to:

```
1 grammar se.lth.cs.jastaddxtext.aspect.Aspect with jbase.Jbase
```

More details for using Jbase with Xtext can be found in an article [26] written by Bettini in 2015.

Bibliography

[1] Lorenzo Bettini. *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013.

[2] MetaBorg Software Foundation. Spoofax, 2006. http://www.metaborg.org/spoofax/meta-language.

[3] Christian Wende, Mirko Seifert, Florian Heidenreich, Sven Karol and Jendrik Johannes. Emftext, 2007. http://www.emftext.org/index.php/EMFText.

[4] The Eclipse Foundation. Xtext, 2009. https://eclipse.org/Xtext/.

[5] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3), 2000.

[6] Niklas Fors, Gustav Cedersjö, and Görel Hedin. Javarag: a java library for reference attribute grammars. In *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015, Fort Collins, CO, USA, March 16 - 19, 2015*, pages 55–67, 2015.

[7] Görel Hedin. An introductory tutorial on jastadd attribute grammars. In *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009, Braga, Portugal, July 6-11, 2009. Revised Papers*, pages 166–200, 2009.

[8] Mark van den Brand, Paul Klint, Jurgen Vinju. The syntax definition formalism sdf, 2007. https://en.wikipedia.org/wiki/Syntax_Definition_Formalism.

[9] Spoofax Language Workbench. Stratego, 2007. http://strategoxt.org/.

[10] The Eclipse Foundation. Eclipse modeling framework - emf, 2009. https://eclipse.org/modeling/emf/.

[11] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Commun. ACM*, 20(11):822–823, November 1977.

[12] ANTLR/Terence Parr . Another tool for language recognition - antlr, 1992. http://www.antlr.org/.

[13] The Eclipse Foundation. Xtend, 2011. http://www.eclipse.org/xtend/.

[14] Eva Magnusson and Görel Hedin. Circular reference attributed grammars - their evaluation and applications. *Sci. Comput. Program.*, 68(1):21–37, 2007.

[15] The Eclipse Foundation. Syntax highlighting in xtext, 2009. http://www.eclipse.org/Xtext/documentation/310_eclipse_support.html.

[16] John Tang Boyland. *Descriptional Composition of Compiler Components*. PhD thesis, 1996. AAI9722877.

[17] The Eclipse Foundation. Xbase, 2010. https://wiki.eclipse.org/Xbase.

[18] Torbjörn Ekman and Görel Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM.

[19] A. Danial. Cloc — count lines of code, 2009. http://cloc.sourceforge.net/.

[20] Lorenzo Bettini, Dietmar Stoll, Markus Völter, and Serano Colameo. Approaches and tools for implementing type systems in xtext. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers*, pages 392–412, 2012.

[21] Lorenzo Bettini. Xsemantics, 2012. http://xsemantics.sourceforge.net/.

[22] Lorenzo Bettini. Xtext type systems (xts), 2011. http://xtypes.sourceforge.net/.

[23] Christoff Bürger, Sven Karol, Christian Wende, and Uwe Aßmann. Reference attribute grammars for metamodel semantics. In *Software Language Engineering - Third International Conference, SLE 2010, Eindhoven, The Netherlands, October 12-13, 2010, Revised Selected Papers*, pages 22–41, 2010.

[24] Guido Wachsmuth, Gabriël D. P. Konat, and Eelco Visser. Language design with the spoofax language workbench. *IEEE Software*, 31(5):35–43, 2014.

[25] Lorenzo Bettini. Jbase, 2015. https://github.com/LorenzoBettini/jbase.

[26] Lorenzo Bettini. Tutorial: Embedded java with xtext, 2015. https://typefox.io/tutorial-how-to-embed-java-in-an-xtext-dsl.

INSTITUTIONEN FÖR DATAVETENSKAP | Lunds Tekniska Högskola | Redovisas 3 JUNI 2016
EXAMENSARBETE: Användning av JavaRAG i en Xtext editor
STUDENT(ER): Emin Gigovic, Philip Malmros
HANDLEDARE: Jesper Öqvist (LTH)
EXAMINATOR: Görel Hedin (LTH)

En kraftfullare snickarbänk för programspråk

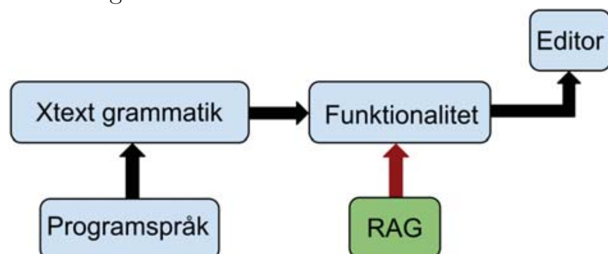
POPULÄRVETENSKAPLIG SAMMANFATTNING AV *Emin Gigovic, Philip Malmros*

ATT IMPLEMENTERA EN EDITOR FÖR ETT PROGRAMMERINGSSPRÅK ÄR EN TIDSKRÄVANDE PROCESS. DET FINNS LYCKLIGTVIS FLERA SÅ KALLADE ”SPRÅKSNICKARBÄNKAR” (LANGUAGE WORKBENCHES) SOM KAN HJÄLPA TILL MED DET HÄR, MEN DE ÄR INTE ALLTID SÅ LÄTTA ATT ANVÄNDA. VÅRT ARBETE HAR FOKUSERAT PÅ ATT INTEGRERA EN AV DESSA SPRÅKSNICKARBÄNKAR MED ETT KRAFTFULLT VERKTYG SOM KALLAS RAGS (REFERENSATTRIBUT-GRAMMATIKER), SOM GÖR DET LÄTTARE ATT IMPLEMENTERA VISSA FUNKTIONER.

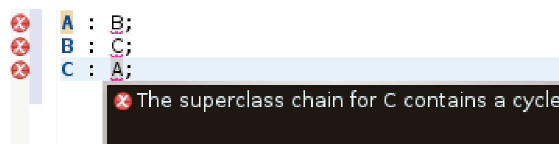
Ett program som används till att implementera en editor för programmeringsspråk kallas ofta för en ”språksnickarbänk”. En enkel jämförelse kan göras mellan ett sådant här program och hur man i en generisk ordbehandlare som Microsoft Word lägger till stöd för olika språk. Vanliga språk som Svenska eller Engelska använder sig av olika ord, har annorlunda grammatik etc., och samma princip gäller för programmeringsspråk. Om ett språk ska ha fullt stöd i en programmerings-editor räcker det dock inte med att språkets grammatik hanteras korrekt, utan det krävs speciella markeringar för vissa ord eller fraser så att de är lättare att urskilja, samt en uppsjö av andra funktioner. På grund av att det är så mycket som ska ingå i en språksnickarbänk har de en tendens att vara ganska komplexa, och att implementera fullt stöd för ett nytt programmeringsspråk är ofta väldigt tidskrävande.

Hur kan RAGs hjälpa till?

Bilden nedan är en väldigt simpel illustration av hur en språksnickarbänk vid namn Xtext fungerar. Första steget är att välja ett programspråk man vill bygga en editor för. Sedan konstruerar man en s.k. grammatik som kan beskriva hela språket. När detta är klart kan man genom Xtext lägga till en massa funktionalitet för de olika delarna av grammatiken.



En av de viktigaste sakerna att lägga till är semantisk analys, som kontrollerar att om något bryter mot språkets grammatik, så ska det markeras på något sätt. Bilden nedanför visar ett exempel på en felkontroll som blir enklare att utföra med RAGs, än med endast Xtext funktionalitet. Att definiera semantisk analys för ett språk är RAGs väldigt bra på, och det vi ville utvärdera var om vi med hjälp av RAGs kan förbättra den analys man normalt har tillgång till i Xtext. För att testa detta implementerade vi tre olika editorer. Två av dem hanterade samma språk, men där den ena bara fick använda sig av standard Xtext funktionalitet, kunde den andra också använda RAGs. Den tredje editorn hanterade ett mer komplext språk än de andra, så att RAGs kunde användas till mer avancerade funktioner.



Resultat

För att integrera Xtext med RAGs använde vi kodbiblioteket JavaRAG, som tillåter användning av RAGs i projekt som är skrivna i språket Java. Då det finns ett visst överlapp mellan funktionaliteten i Xtext och RAGs var ett av de huvudsakliga problemen att bestämma när man skulle utnyttja RAGs istället för standardfunktioner från Xtext. Till slut drog vi slutsatsen att RAGs främst var användbart när man behövde mer avancerad funktionalitet, och särskilt då man ville komma åt svårtillgänglig information vid implementation av semantisk analys. Vi har genom vårt arbete visat att det är möjligt att integrera Xtext med RAGs för att lösa problem som kan vara svåra att lösa med enbart Xtext, vilket kan vara användbart i kommande Xtext projekt.