Parsing by Example

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von

Markus Kobel

März 2005

Leiter der Arbeit:

Prof. Dr. Oscar Nierstrasz
Prof. Dr. Horst Bunke
Tudor Gîrba
Prof. Dr. Michele Lanza

Institut für Informatik und angewandte Mathematik

The address of the author:

Markus Kobel
Rosenweg 4
CH-3303 Jegenstorf
`kobel@acm.org`
`http://www.iam.unibe.ch/~kobel/`

Abstract

We live in a world where we are surrounded with information technology. The software systems around us are countless. All of those systems have been written once and must be maintained today. While a system evolves it becomes difficult to maintain. We use reengineering tools today to simplify maintenance tasks. With the support of such tools we can change the form of software systems in a way that makes them easier to analyze.

Before we can use any reengineering tool with a software system we must reverse engineer that system. To reverse engineer a software system means that we need to build a model from the system. This model represents our system in a more abstract way than the source code itself does. The way from the source code to the model is often a problem. If a reengineering tool supports a specific model the maintainers of that tool must provide a parser for every programming language they want to support. Such parsers translate source code written in a particular language into a model. There are so many languages used in systems today that it is not possible to support all of them. Additionally, the languages themselves evolve and so we need parsers for every version and every dialect of a programming language. There are a number of approaches to solve that problem (for example fuzzy parsing). Most of these approaches are not flexible enough for today's needs: We cannot adapt them to another programming language or if we can we need a lot of knowledge about the language and about the whole parsing technique. Depending on the technique that we use we must write a parser or at least a grammar as a basis for a parser generator. In most of the cases this is a difficult and time-consuming task.

Our idea is to build an application that generates parsers based on mapping examples. A mapping example is a section in the source code to which we assign an element in our target model. Based on these examples, our application builds grammars and generates a parser. If the parser fails to parse some code our application asks the user to provide more examples. This approach is flexible enough to work with a software system written in an arbitrary programming language. The user does not need to have knowledge about parsing. However, he should be able to recognize the elements in the source code that he wants to map on the elements in the target model.

We prove the flexibility of that approach with our reference implementation called *CodeSnooper*. This application works with any input. As target model we take the FAMIX model that is used by the MOOSE reengineering environment.

Acknowledgements

First of all, I want to thank Prof. Dr. Oscar Nierstrasz for giving me the possibility to do this work in his group. The subject of this work was in his mind for a long time now. I always felt that he believes in my work during the last year. Thanks go also to Tudor Gîrba who supervised this work. With his suggestions, CodeSnooper became a usable tool.

I also want to thank Prof. Dr. Horst Bunke for his uncomplicated supervision. I often forgot to inform him about the state of this work - even so, he was never angry.

Special thanks go to Prof. Dr. Michele Lanza. Even though he went away from Berne I always knew that I can reach him if it was needed. I want to thank you, Michele, for all you have done for me - first as a teacher, then as my boss and supervisor and in the end as a friend.

There are a lot of students i got to know during the last few years. If I want to name all of them this list becomes endless and I would forget someone for sure - so, I just thank all of you for providing a pleasant atmosphere during my studies. Special thanks go to the students that shared many days in the lab with me: Thomas Bühler, Adrian Lienhard and Michael Meer. Another group of (former) students I want to thank are the legendary *bitdefenders* composed of Michael Locher, Mauricio Seeberger, Christoph Hofer and Stefan Leuenberger. Thank you for the endless discussions and coffee klatsch... ;)

Many thanks go to my whole family. A special place in my life have my parents - they supported my will in the last years although it wasn't easy. My sister and my brother do have a special place, too. Thank you for sharing your youth with me. I know that my mind often went in another direction than yours.

There is one last person that I want to mention by name. It's my godfather Walter Lüthi - thank you for your support and your belief in my work. I know you often had to wait a long time...

Last but not least I want to thank all of my friends that I got to know before and beside my studies. Unfortunately not all friendships survived my studies - but the otherones are stronger than before.

If you find a solution and become attached to it,
the solution may become your next problem.

Markus Kobel
March 2005

Contents

Chapter 1

Introduction

"A program that is used in a real-world environment must change,
or become progressively less useful in that environment.
As a program evolves, it becomes more complex, and extra resources
are needed to preserve and simplify its structure." [LEHM 85]

The field of software reengineering becomes more and more important. But we can only reengineer a system when we understand its structure. In order to understand that structure we have to reverse engineer that system. There are several approaches to reverse engineer a program. Among them are the following approaches: reading existing documentation and source code, running the software and/or generating and analyzing execution traces, interviewing the users and developers, using various tools, analyzing the version history, et cetera [DEME 02].

A tool that supports us in getting an overview of a software system must somehow translate that system into a model. This translation is a challenging point. Someone must write a parser that can translate that software system into the model he wants to support. So, the maintainers of such tools must provide a parser for every programming language they want to support. But it is not only the number of languages that is a problem. A language itself also evolves. A parser that works with a specific version and/or dialect could not work with the next version anymore.

A parser does two things while processing its input:

1. Split the input into tokens.
2. Find the hierarchical structure of the input.

First we have a *lexical analyzer* (scanner) that splits the input into tokens (point 1). The *syntax analyzer* takes these tokens as input and generates a *syntax tree* (point 2). To simplify the task of writing such analyzers Lesk and Johnson published papers on *Lex* (a lexical analyzer generator) [LESK 75] and *Yacc* (yet another compiler-compiler) [JOHN 75]. There are different implementations of *Lex* and *Yacc* available today. For example flex[1] and bison[2] from the GNU project[3].

Lex source is a table of regular expressions and corresponding program fragments. This table is translated into a program that reads an input stream, copying it to an output stream and splitting the input into strings which match the given expressions. The program fragments are executed in the order in which the corresponding regular expressions occur in the input stream.

[1]`http://directory.fsf.org/flex.html`
[2]`http://directory.fsf.org/bison.html`
[3]`http://www.gnu.org/`

Yacc takes as input the grammar of a language. Each production in that grammar matches a specific structure of the language. The user of *Yacc* specifies an action for each production. *Yacc* converts that grammar into a program that reads an input stream and whenever a structure is found it executes the action of the corresponding production.

Both *Lex* and *Yacc* produce C code. There are also similar tools available for other programming languages. When speaking about the input of a lexical analyzer generator we use the term *scanner definition*. We use a scanner definition together with a grammar as input for a parser generator.

It is easier to write a scanner definition and a grammar than a parser. But writing a grammar is still a time-consuming and difficult task: This is a problem if we want to parse a lot of different inputs.

There are several approaches known today that address this problem. Often we do not need to fully parse a system to have a valid model. There are several approaches that extract a partial model from the input.

One approach is *Fuzzy Parsing*: the idea is that we have some anchor terminals and based on them we can extract a partial source code model [KLUS 03]. We can also build a parser based on a *Island Grammar* where we define some productions for islands (things we want to recognize in the source code) and every other production leads to water (things we do not care about) [MOON 01]. We also know about *Generalized Parsing*: such a parser can handle ambiguities in the provided grammar and follows every possibility to parse the input [VAN 98]. There are also parser generators like *DURA* that produce parsers with additional features like backtracking [BLAS 01]. No matter which approach we use, we must have a good knowledge about the programming language and about the parsing technique. The problem we are addressing in this thesis is how to provide a useful grammar for a parser generator.

Our target is to generate a model based on provided mapping examples. A mapping example is a section of some source code to which we assign an element in our target model. Based on such examples we build grammars and with them we generate parsers. With that approach we can get a model from any input in a short time period and without a deep knowledge about parsing techniques. However, we must be able to recognize the elements we want to detect in the given source code.

The following points are the core requirements that we want the reference implementation of our approach to fulfill:

- **Reliability**: The application must work with any input without crashing. In the worst case it does nothing at all. If it can only parse a part of the system it must provide a valid model from the successfully parsed part.
- **Exception handling**: If the generated parser fails to parse the input we want enough feedback to know why this happened. If it does not know how to parse part of the input it must ask us to provide another example.
- **Incremental parsing**: We want to build up models in an iterative and incremental way. As an example for an object-oriented system, we just detect *Classes* in a first run and then, we add *Methods* and *Attributes* to that model in a second run.
- **User interface**: We want a user interface where we can specify mapping examples in an intuitive way.

1.1 Our Approach: Parsing based on Examples

We need a model of the system we want to analyze. In most cases we do not need to fully parse that system. In a first step it is often enough if we can extract a partial model of the system. For example in an object-oriented system that could be *Classes* in a first step and *Methods* and *Attributes* in a second iteration.

We specify only examples of the mapping between our input and elements of a target model. Based on this information we generate a parser that can translate the whole input into a model. When generating the parser we want to generalize the given examples in order that the parser can detect other instances of these target elements as well. If there are sections in the source code that the parser cannot recognize we specify some more examples. We use this additional information together with the initial examples as basis for the next parser. With this approach we extract information from a system written in any language.

If we have a system that is written in a language that we do not know, usually we do not have a precise parser that can give us a model for that system. When we analyze such a system and recognize some instances of the elements in our target model we can specify them as examples: We can extract a model from a system without knowing the language.

We validate our approach with the support of our reference implementation called *CodeSnooper*. To work with *CodeSnooper* the user does not need any knowledge about parsing. *CodeSnooper* works with any input, generates grammars and, based on them, parsers. It gives feedback if a parser fails to parse any part of the input.

With our approach we mix precise parsing with elements from fuzzy parsing and island grammars. We do not just skip input until we find a specific anchor as is done in fuzzy parsing. We only ignore certain tokens in well defined contexts. We can say that the grammar we generate is some kind of island grammar. Since we generate our grammars based on examples we can get more complex grammars in less time compared to writing them by hand.

The parser that we get from our compiler-compiler is a LALR(1) parser - that means it is a **L**ook **A**head **L**eft to right parser and produces a **R**ightmost derivation with a look ahead of **1** input symbol. There is a large set of context-free grammars from which we can generate LALR(1) parsers. LR parsers are commonly used by programming language compilers and therefore used for this work.

In Figure 1.1 we see how we arrive from source code to a model: We generate a *Node* from our input (source code). We specify examples of the mapping between source code and our target model directly on this *Node*. This *Node* becomes a tree because we split it up and add subnodes when specifying examples. This tree is explained in Section 3.3. We have two possibilities to get a model from that tree:

1. We export the example tree directly as a model. The model contains exactly the elements that we specified as examples.
2. We generate a grammar from our examples. We give this grammar to the compiler-compiler and generate a parser that we use for parsing the whole source code. The parse tree that we get is built with the same *Nodes* as our examples. We export this parse tree as a model.

The way from the source code to a model is explained in detail in Chapter 3.

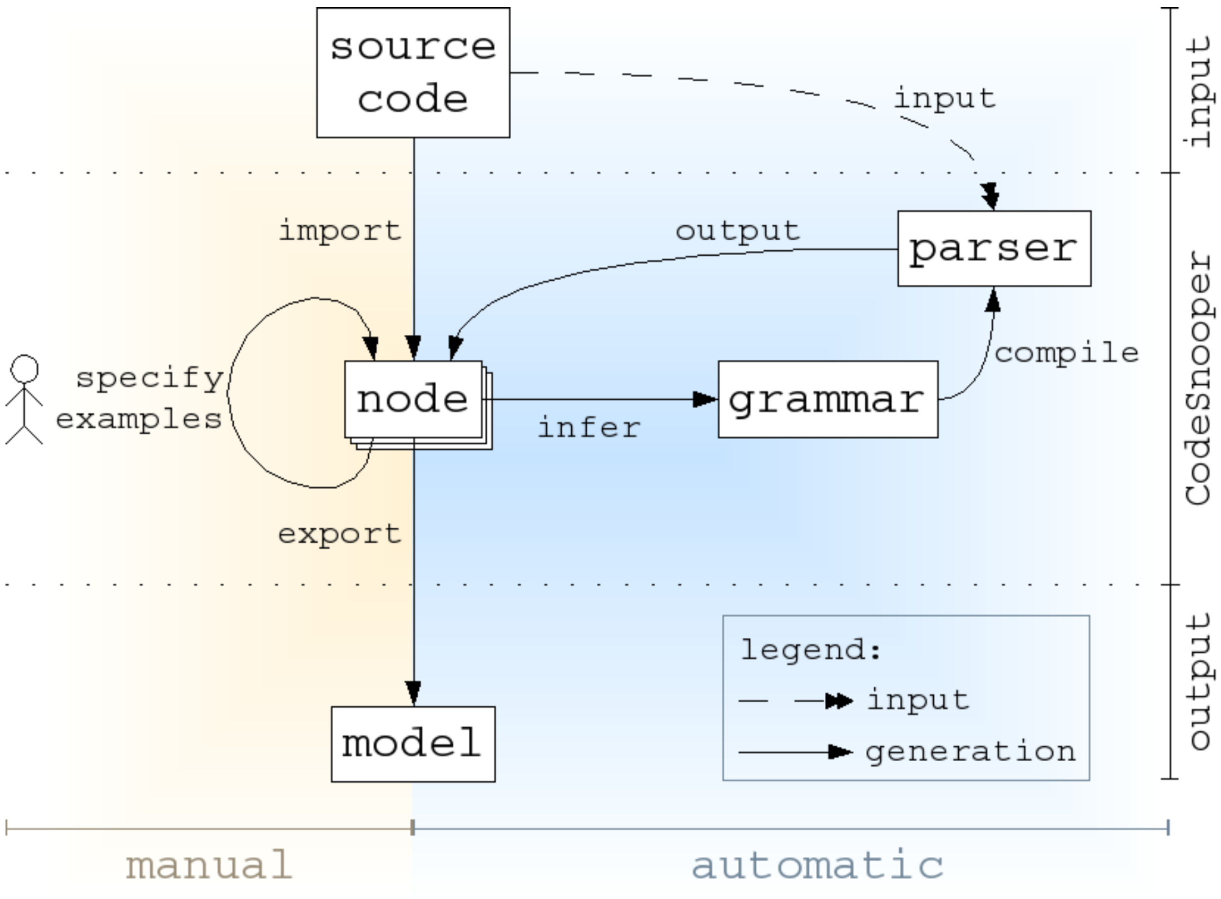


Figure 1.1: The way from source code to a model.

With such a design we can be sure to get a raw model from any software system. Even if our generated parser is not able to parse a whole system we can use that application as a direct interface between any source code and our target model - at least, we can always translate the given examples into a valid model.

1.2 Goals of this Work

Here are some questions that we would like to answer in this work:

- Is it possible to build up an application that can generate grammars/parsers on the fly based on some mapping examples?
- What information is needed from the user at least to build a usable grammar?
- How many examples do we need to build a usable grammar?
- How much time do we need to make enough examples for building up a usable grammar?
- Is there information in the source code that we cannot transform into a model with our approach?
- What does a useful user interface for such a tool look like?

1.3 Structure of this Document

This document consists of the following chapters:

- In Chapter 2 we discuss the parsing techniques that are used in reengineering environments at the moment. We investigate a few approaches and their assets and drawbacks. We take a look at the problems that these approaches cannot solve.
- In Chapter 3 we present our approach '*Parsing by example*'. We discuss every step that is needed to parse a system with our approach. We look at the specification of examples and the inference of grammar rules based on these examples. We also discuss all the components of our grammar. We explain how we build our parse tree and how we can export a model based on such a tree. Finally, we show how we can store a complete parser definition.
- In Chapter 4 we summarize experiences that we made during the work with our application. We take two examples of source code that we transformed into a model with our approach and compare them with the models that we get while using a conventional method.
- In Chapter 5 we evaluate our approach. We discuss the problems we had and provide possible solutions. We also look at the limitations of our approach.
- Chapter 6 contains a summary of our work and experiences we gained during this work. We compare our approach with the other ones. We discuss if we can solve the problems that other approaches cannot solve. We also look at possible future work.
- Appendix A contains an overview of the application that we implemented during this work. This tool is named CodeSnooper and we used it as a proof of concept for our approach. We look at the requirements that we had and explain the functionality that was needed in order to get a usable tool.
- In Appendix B we provide the document type definition for our parser definitions, give an overview of our validation environment and include an example of a complete grammar.

Chapter 2

State of the Art

2.1 Parsing for Reverse Engineering

Today, there are many software systems that are difficult to maintain because they evolved over time. We use reengineering tools to simplify maintenance tasks. Often, there is a problem to transform these systems into a model that can be used by a reengineering tool. There is a large number of programming languages and dialects. One would need a parser for every single language and its dialects. Because of that almost every reengineering environment provides an external representation of the model it uses in one or more common file formats. For example the MOOSE reengineering environment supports the import from CDIF and from XMI. Source code written in C/C++, Java or Ada can be parsed and outputted in the CDIF format by Sniff+[1]. With such external parsers the maintainers of a tool can extend its support for different languages. The drawback is that they must rely on external tools.

The maintainers of reengineering frameworks must provide a parser for every programming language they want to support with their tools. For every language (or even dialects) they must generate a parser or use an external one. But it is not the principal task of a maintainer to write parsers all the time. It may be that external parsers don't really provide all the information that are needed - so, it would often be an advantage if they could provide a parser that the user could tune while working with it.

2.2 Different Approaches

There are different levels of parsing. They go from 'Lexical Analysis' to 'Precise Parsing'. Between these two barriers there is 'Fuzzy Parsing', 'Island Grammars', 'Skeleton Grammars' and 'Error Repair' [KLUS 03]. Beside these parsing techniques there are also some quite different approaches described and/or implemented. We look at some of these different parsing approaches more closely in the next sections.

2.2.1 Fuzzy Parsing

Most reengineering frameworks use a form of fuzzy parsing in order to support more programming languages or more dialects of the same programming language. The goal of a fuzzy parser is the extraction of a partial source code model based on a syntactical analysis. The key idea of fuzzy parsing is that there are some anchor terminals. The parser skips all input until an anchor terminal is found and then context-free analysis is attempted using a production starting with the found anchor terminal [KLUS 03].

[1]`http://www.windriver.com/products/development_tools/ide/sniff_plus/`

2.2.2 Island Grammars

With island grammars we get tolerant parsers. An island grammar is a grammar that consists of detailed productions describing certain constructs of interest (the islands) and liberal productions that catch the remainder (the water). By varying the amount and details in productions for the constructs of interest, we can trade off accuracy, completeness and development speed. There are some different versions of island grammars known besides the one that we just defined [MOON 01]. Leon Moonen speaks of the following:

- **Lake grammar**: When we start with a complete grammar of a language and extend it with a number of liberal productions (water) we get a *lake grammar*. Such a grammar is useful when we want to allow arbitrary embedded code in the program we want to process.

- **Islands with lakes**: This is a mix of productions for islands and water. We can specify nested constructs as islands with lakes.

- **Lakes with islands**: This is an other mix of productions for islands and water. We can for example detect things in an embedded extension with an island in a lake.

2.2.3 RegReg

RegReg is a generator of robust parsers for irregular languages. The generated parsers are based on a cascade of lexers. Each lexer acts at a certain level and uses as input the stream of tokens produced by the lexer one level above: Level 1 deals only with characters, level 2 is based on tokens produced by level 1 and level 3 is based on tokens from level 2. There is no limit set on the number of levels although at least one level is required.

There are some studies available on parsing code using regular expressions. They show that there are limitations with such an approach if they want to do precise parsing as for example with an LR parser. This is not the intended goal of RegReg. The developers of RegReg just want to do a partial parsing like they can do with fuzzy parsing or island grammars. Mario Latendresse analyzed case studies about fuzzy and island parsing and showed how he can use RegReg to solve the studies addressed by the other approaches [LATE 03].

2.2.4 Generalized Parsing

A generalized LR parser (GLR) is a generalization of a LR parser. The GLR algorithm uses an ordinary LR parse table, but this table may contain shift/reduce and reduce/reduce conflicts. If there is a conflict encountered while processing the input the GLR algorithm forks as many parsers as there are possibilities to proceed: The various possible evolutions of the LR stack are maintained simultaneously. If the conflict was due to the need of lookahead, the forked parsers die. These parsers proceed in parallel and synchronize on *shift* actions: Parsers that are in the same LR state are merged together.

The results are parse bushes or forests opposed to plain parse trees. However, for reverse engineering purposes we are only interested in one tree. The number of trees is reduced by applying a numer of syntactic disambiguation rules. If there is more than one tree left over in the end, the user must make a selection. This approach is based on the optimistic assumption that large parts of the input can be analyzed with a plain LR parser without the requirement to clone LR stacks [VAN 98].

2.2.5 DURA

DURA is the name of a tool designed and implemented by Darius Blasband for his PhD thesis. DURA is the kernel technology behind RainCode[2]. RainCode is a generic source code manipulation tool. This technology has proved its usefulness in many industrial projects with several millions lines of code written in different programming languages (COBOL, PL/1, C & C++, Java, ...). Unfortunately RainCode is not an open project.

LexYt is a lexical analyzer generator, similar to YACC's lex with some additional features (backtracking lexers, ...). The generated lexers can be coupled with parsers produced by DURA. Compared with a plain LR parser, DURA-generated parsers provide an 'undo' operation beside 'shift' and 'reduce'. Based on the last performed operation this is an 'unshift' or an 'unreduce'. DURA takes the optimistic view from GLR parsing further: Not only can a plain LR parser handle most of the input, but in case of conflicts, it must not to go very far to bump into a problem if it selects the wrong path [BLAS 01].

2.3 Problems with Different Approaches

Almost every approach does have some assets and some drawbacks. We want to mention a few problems of different approaches here:

- **Generalization is missing:** This is a common problem in almost every approach. In the end there is a parser that works with a certain family of programming languages but not with an arbitrary language. We can use any approach to solve a specific problem. We can write the correct grammar and generate a parser based on it. Users and also maintainers of reengineering environments are not here to write grammars/parsers. They need support from a tool for this task. If we want to import a system written in any programming language into a reengineering environment we do not want to write a parser and learn all the things that are needed for that. We need an easy to use application that helps us to transform our system into our target model within a short time period. None of the approaches we know provide such support for the end user.
- **No error recovery:** If the parsing of a system fails most approaches do not help the user to recover from the error. Mostly we do not even get feedback why and where there were problems.
- **Context is not taken into account:** Fuzzy parsing is an approach that never cares about the context of structures of interest. There are many cases where we want to work with the context of a detected element.

[2]`http://www.raincode.com/`

Chapter 3

Parsing by Example

Our goal is to model a software system. This transformation is based on mapping examples. A mapping example is a section of the given input for which we define a corresponding element in our target model. In Figure 3.1 we see how we get from some input to a valid model. We repeat steps 2 to 5 until we are satisfied with the result. In this chapter we go into the details of these steps and answer the following questions:

1. *We need a number of input files.*
2. *We specify mapping examples that we find in the input.*
   How do we specify examples? (Answer: see Section 3.2)
   How do we store the information that we get from the user? (Answer: see Section 3.3)
3. *We derive a grammar from the mapping examples.*
   How do we generate grammars from the given examples? (Answer: see Section 3.6)
4. *We generate a parser based on the derived grammar. The parser needs also a scanner that splits up the input into tokens. Thus, we need also a scanner definition beside the grammar.*
   What does the scanner look like? (Answer: see Section 3.1)
   What components do we use in a grammar? (Answer: see Section 3.4)
   How do we generate a parser from the scanner definition and grammar? (Answer: see Section 3.7)
5. *The parser is used to transform the input files into parse trees. If it fails to parse any input, the parser gives feedback to the user.*
   What is the output of our parser when it parses some source code? (Answer: see Section 3.8)
6. *In the end we can export all parse trees as a model.*
   How do we export our parsed code as a model? (Answer: see Section 3.9)
7. *We need to store the parser definition for later use.*
   How do we store a parser definition? (Answer: see Section 3.10)

We use many examples to illustrate our approach. All of them are written in Java because Java is well-known to people working in the field of software engineering. Most of the examples are reduced to the significant parts.

We also give some examples of grammars and scanner definitions. Because such definitions depend on the parser generater that we use we provide some information about the implementation here:

For the reference implementation of our approach (*CodeSnooper*) we work with the FAMIX model [DEME 01, TICH 00] as target model and with MOOSE [DUCA 05] as reengineering tool. We use

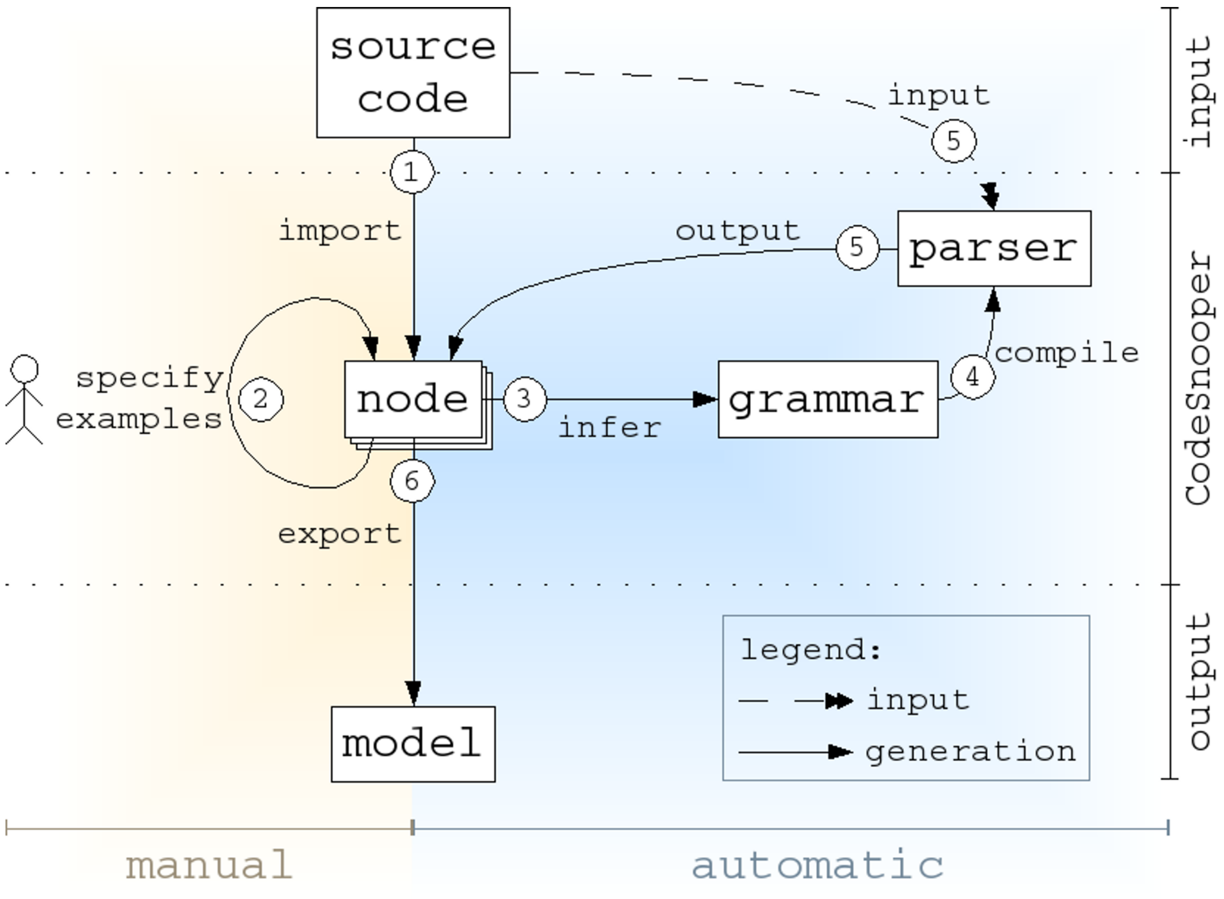


Figure 3.1: The way from source code to a model - step by step.

Cincom Smalltalk[1] as programming language because the whole MOOSE reengineering environment is developed within this language. As parser generator we use the Smalltalk Compiler-Compiler (SmaCC[2]). We have to respect the grammar and scanner definition syntax used by SmaCC. We also use this syntax for our examples of grammars and scanner definitions in this work.

3.1 Scanner

Many parsers have a scanner attached that splits the input into tokens. The parser can then make its decisions based on these tokens and must not work with simple characters.

A simple scanner token is expressed as a string. A more complex one is defined with a regular expression. With SmaCC it is possible to define simple tokens in the grammar itself without having a scanner token for it. For example if we want to handle the character "{" in a special way we have two possibilities:

- We define a scanner token that looks like: `<left_curly_brace> : { ;`
  Then we can use this token in our grammar productions.
- We can use the character directly in our grammar productions if we put double quotes around it. The generated parser then recognizes it as a token and handles it in the same way as it would have been defined in the scanner.

[1]http://smalltalk.cincom.com/
[2]http://www.refactory.com/Software/SmaCC/

So, we define only a few more complex scanner tokens. We put all the other tokens we have to handle directly into the grammar productions. In the end there are only a few scanner tokens that we rely on. The following example shows the scanner definition that we use for parsing source code written in Java.

```
<DECIMAL_INTEGER>:   0 | [1-9] [0-9]*    ;
<HEX_INTEGER>:       0[xX][0-9a-fA-F]+   ;
<OCTAL_INTEGER>:     0[0-7]+             ;
<IDENTIFIER>:        [a-zA-Z_$] \w*      ;
<eol>:               \r
                   | \n
                   | \r\n                ;
<comment>:           \/\/ [^\r\n]* <eol>
                   | \/ \* [^*]* \*+ ([^/*][^*]* \*+)* \/ ;
<whitespaces>:       [ \f\t\v]+          ;
```

If we want to work with another programming language, we only have to change two scanner tokens: `<IDENTIFIER>` and `<comment>`. For example in Ruby there are question marks allowed in identifiers, so we have to take that into account. We can also change other tokens but they are the same for many languages.

There are many characters that such a scanner cannot recognize. We handle all special characters directly in the grammar. We call them *LanguageConstructs* (described in Section 3.5).

3.2 Mapping Examples

Every example we make points to a special target element in our model. We have to specify the whole signature in the source code that is needed to define this target element in the used programming language.

As an example we take the following Java class:

```
class AJavaClass {
    public void hello() {
        System.out.println("Hello World!");
    }
}
```

The signature for a *Class* can be reduced to:

```
class AJavaClass { <not_known> }
```

This information alone is not enough. We do also need to know that "**class**", "{" and "}" are keywords/tokens of the language and "**AJavaClass**" is the name of that entity. For the signature we do not care about the source code in the body of the *Class* at all. So, we just say that this code does not belong to that entity definition (<**not_known**>).

Another signature could be the one for *Method*. For this example that would be:

```
public void hello() { <not_known> }
```

Here we have to know that "**public**", "**void**", "(", ")", "{" and "}" are keywords/tokens and that "**hello**" is the name of that entity. As we did it for a the *Class* above we do not care about the body of the *Method* either (<**not_known**>).

Each example is represented by a *Node* in a tree. For a detailed description of the tree see Section 3.3. Within these examples we take care of the hierarchy that evolves while specifying the signatures: If we define the signature of a *Method* within the signature of a *Class* this leads to a structure where a *Method* is a child of a *Class*. This is represented in our tree so that this *Method*-example is a child-*Node* of the *Class*-example-*Node*.

3.3 Example Tree

When specifying examples we build a tree of these examples. Each *Node* has the following properties:

- It knows its parent node.
- It knows its child nodes.
- It knows its target element in the model. It is possible that a node does not have a target element in the model. This means that we want to ignore the contents of that node while parsing.
- Most elements in the target model have some attributes. The node also knows which of them are set for itself.
- It knows its name unless there is no target specified. Every entity that has a target attached must have a name. We use this name as a unique name for identifying this entity in the exported model.
- It knows its original source code.
- If a node has child nodes it also knows from which part of its own source code they are generated.

We look at the tree that we get from the examples of Section 3.2 a bit more closely. In Figure 3.2 we can see the simple tree that we build while specifying examples. Every *Node* knows its parent and its children. They also know about their target and their name. In Figure 3.3 we see an inside view of our nodes with the source code that is attached to the nodes. As shown in the figure, there is a component in the *Node* that manages the source code. This component navigates to the subnode(s) of its enclosing node. This is important when generating grammar rules from such a tree. In Section 3.6 we describe how we use this information in order to get a grammar rule with the appropriate nonterminals in the proper place.

3.4 Components of a Grammar

Figure 3.4 shows a class diagram of our *Grammar* and its sourrounding classes. A *Grammar* object consists of the following components:

- It has a list of *GrammarRules*. *GrammarRules* have the following attributes:
  - a unique symbol as its nonterminal.
  - a list of *GrammarProductions*. A *GrammarProduction* knows also the reduce action for its production.

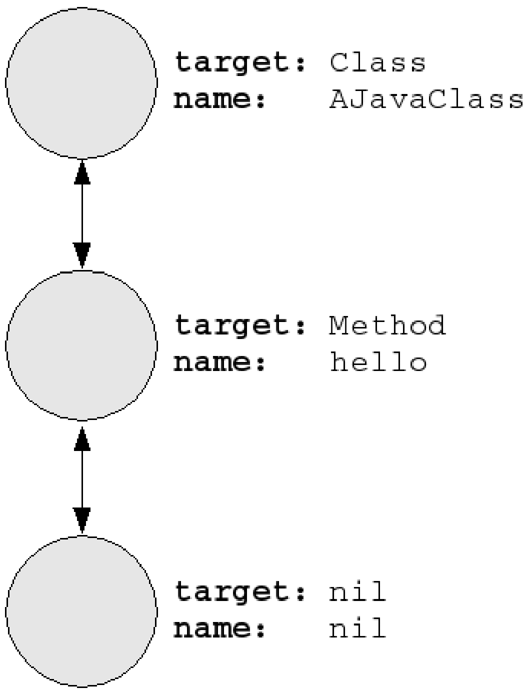


Figure 3.2: External view of example nodes.

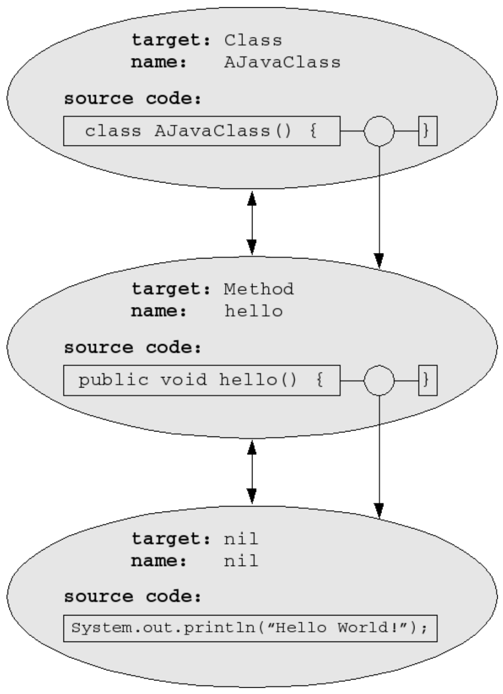


Figure 3.3: Internal view of example nodes.

- It has a list of *LanguageConstructs*. These are all keywords and tokens of the programming language we want to parse unless they are already defined in the scanner. For example in Java this would be the keyword "**class**". A *LanguageConstruct* may also be a block construct. For example the construct "{ **...** }" is an instance of *LanguageConstruct*, too. These *LanguageConstructs* also know in which contexts they should be ignored. For example, we do not care about brackets in the body of a method. This means that the language construct "(**...**)" knows that we do not want it to be recognized as a token in the context of a method. We provide more information about *LanguageConstructs* in Section 3.5.

We export such a *Grammar* object as a grammar that is usable by SmaCC. Our internal representation of a grammar is independent of SmaCC. We did not use the representation that SmaCC uses because this solution would not be flexible enough. We also include more information in our grammar. For example our grammar knows about language constructs.

3.5 Language Constructs

In our approach we do not distinguish between keywords of the programming language and tokens. For example in Java we handle the keyword "**class**" in the same way as the special character "**=**". We manage all of them as *LanguageConstructs*. A *LanguageConstruct* can also be a simple block construct. That is an element of the language that starts with a token and ends with a token. For example an expression that is put in parentheses can be a block constructs. All of these *LanguageConstructs* know in which contexts we want to ignore them. This information is used when exporting a grammar to SmaCC.

If we look at the scanner definition in Section 3.1 we notice that there is no token available that matches a special character. We handle all those characters directly in the grammar. If we put double quotes around a string in a grammar production SmaCC handles this string as a token.

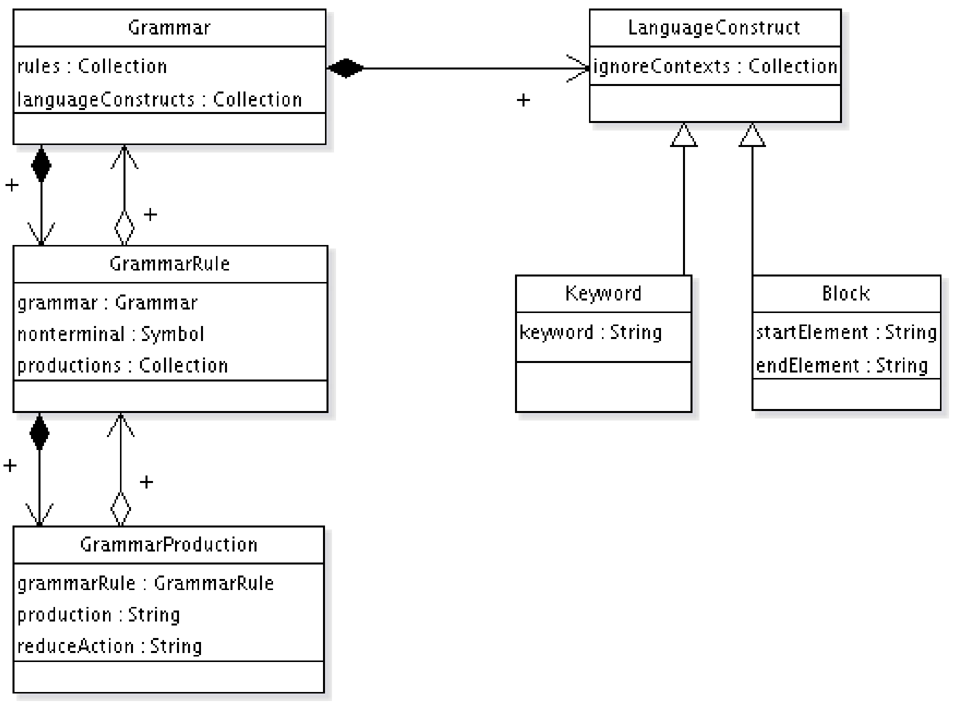


Figure 3.4: Class diagram of our grammar classes.

3.6 Generation of a Grammar

Based on the signatures (see Section 3.2) that we defined we can generate a grammar. These examples are stored in a tree. We traverse this tree and generate a grammar production for each node. If we take the example from Section 3.2 we get the following grammar rules:

```
Class  ::= "class" <IDENTIFIER> "{" Method* "}" ;
Method ::= "public" "void" <IDENTIFIER> "(" ")" "{" not_known* "}" ;
```

We build a grammar rule step by step with the assumption that we already have a grammar where we add our production. The generation of a grammar production based on one node consists of the following steps:

1. We need a nonterminal for this production. We take the name of the target element of this node. For the *Class* example this would be just **Class**. We ask our grammar if there is already a production defined with that nonterminal. If there is one, we take this production, otherwise we create a new one with that nonterminal.

2. We build the production itself. For that, we replace the name of the entity with an identifier token from our scanner (`<IDENTIFIER>`). We put double quotes around all keywords/tokens so that SmaCC recognizes them as tokens of that language. For a description of the scanner see Section 3.1.

3. We go through the signature and look where subnodes are defined. For every subnode we include the name of target of the subnode in the production. It is a special case if the subnode does not have a target. From such nodes we generate *exclude rules*. We look at that in more detail in the next subsection.

If we do these steps for every node in the tree that knows a target, we get a consistent grammar where all used nonterminals are defined.

3.6.1 Nodes without Target

Our *Nodes* may have no target element attached: There is no element in the target model that this node could be mapped to. It may also be the case that we do not want a special node to be mapped to anything (although there would be an element in the target model) - so we do not attach a target to that node.

A *Node* that does not have a target element attached means that its contents should be ignored in its context.

As an example we look at the following Java class:

```
class AJavaClass {
    private String hello = "Hello World!";
    public void hello() {
        System.out.println(this.hello);
    }
}
```

We want to detect *Classes* and *Methods*: We specify the signature of the *Class* and the signature of the *Method*. We exclude the remaining parts of the source code from our signature definitions: These *Nodes* have no target. This leads to the tree of example *Nodes* shown in Figure 3.5.

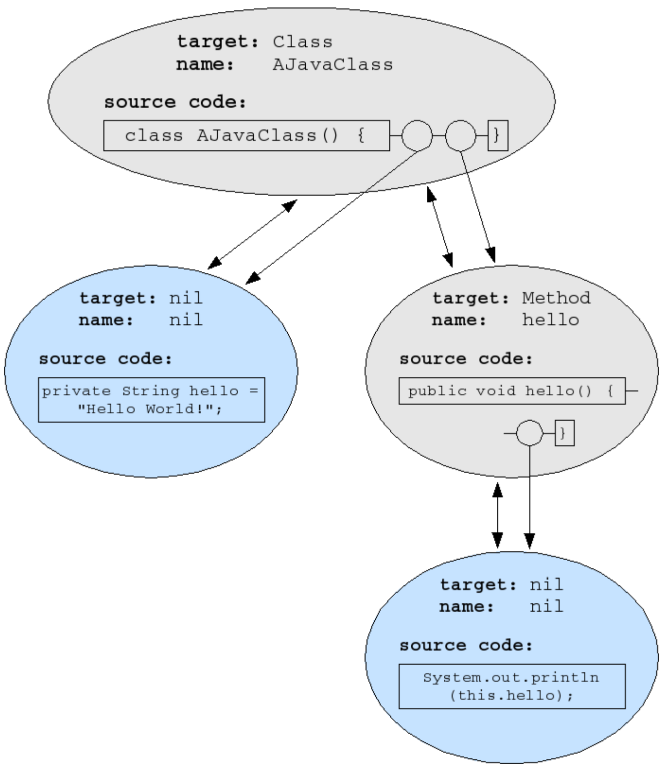


Figure 3.5: Some nodes without a target.

When we follow our rules for the grammar generation we get the following grammar:

```
Class  ::= "class" <IDENTIFIER> "{" (not_known | Method)* "}" ;
Method ::= "public" "void" <IDENTIFIER> "(" ")" "{" not_known* "}" ;
```

The production for *Class* looks different now. According to this production we detect *Methods* in the body of a class and ignore all other things (not_known).

The generation of the grammar becomes more complex if there are some nodes without a target that have subnodes with a target. If we extend the example with a second attribute it could look like that:

```
class AJavaClass {
    private String hello = "Hello";
    public void hello() {
        System.out.println(this.hello + " " + this.world + "!");
    }
    private String world = "World";
}
```

In Figure 3.6 we see one possibility of the tree that we can get when specifying examples. This tree leads to exactly the same grammar as we had before. We can make a transformation of that tree where the result only contains leaf nodes without a target: We split the node without a target into two separate nodes and attach both of them directly to the class node. We also attach the method node that was below that node directly to the class node. So, the root node has now three direct subnodes and the generation of the grammar works without problems.

In our implementation we do not transform the tree. When generating the grammar rules we traverse it in a way that leads to the same result.

It is common to have intermediate nodes without target. There are some nodes that we generate automatically. For example we create nodes from ‘end of line’ tokens. Such nodes do not have a target. We do not show them in the examples because they do not add useful information.

3.6.2 Reduce Actions

Some of the *GrammarProductions* do have a reduce action attached. The productions without a reduce action just return the source code whenever they match. There is one main reduce action that we use all-over in our grammars. It is of the form:

```
makeNodeNamed: <NAME> fromNodes: nodes rule: #<TARGET> production: <NUMBER>
```

Whenever this reduce action is invoked, our parser generates a new *Node* in our parse tree. The new *Node* has the following properties:

- `<NAME>` is the name of the *Node*.
- We search in our list of possible targets the one with the name `#<TARGET>` and attach this target to the *Node*.
- With `#<TARGET>` and `<NUMBER>` we can reconstruct which *GrammarProduction* matched. So, we also attach that production to the new *Node*.
- `nodes` is an collection of subnodes recognized before. These may be instances of *Node* generated by our parser or just strings for all productions without a reduce action. From these `nodes` we can reconstruct the source code for the new *Node* and put already recognized subnodes in the right place.

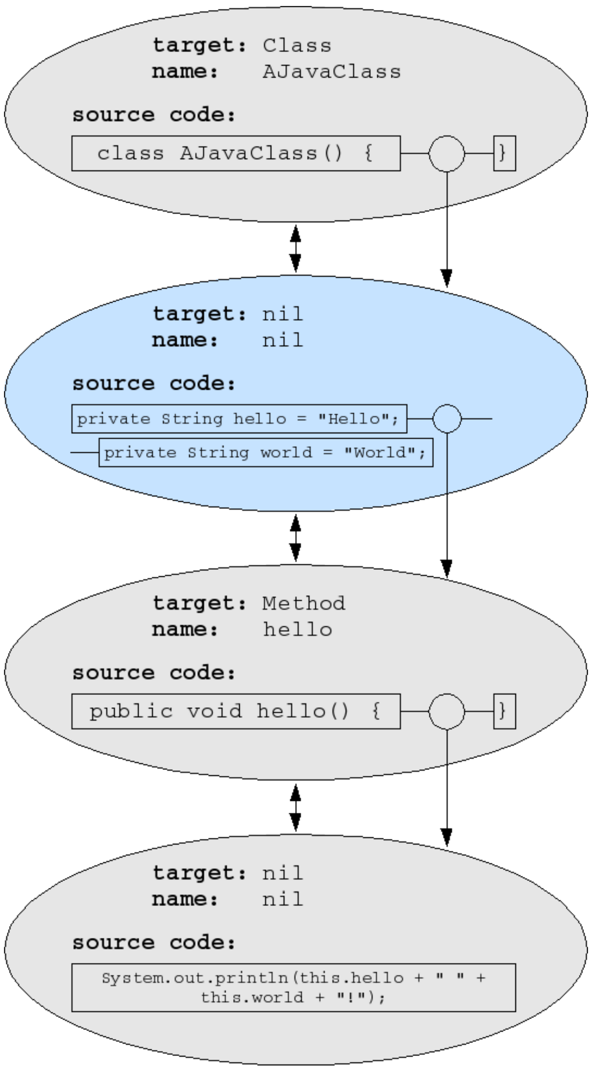


Figure 3.6: Node without a target that has subnodes with a target.

3.6.3 Whitespace

We want to take care of all whitespace characters because we want to be able to restore the look of the source code we work with. Our grammar rules become more complicated when we include whitespace characters. The benefit is that we leave the layout of the code unchanged while parsing. This is worth the effort.

In most programming languages whitespace does not matter: Most parsers do not care about whitespace and already the scanner throws such characters away. There are many cases where nobody reads the parsed code again (for example compiled code). But for us it is interesting to work with the parsed code. For the generation of our grammar we follow a simple rule: If we find a whitespace character in the signature of an example we assume that there may also be none or more than one such character. In other words, we make whitespace optional but nevertheless include it in our parse tree.

If we want to parse source code written in a language where whitespace does matter (for example Fortran 77) we can change the algorithm that we use for the generation of the grammar rules. In *CodeSnooper* we have not included the possibility to change the handling of whitespace for the user because it does not matter in most common programming languages.

3.6.4 Merge Productions

For generating our grammars we made an assumption:

Every example of a specific target element leads to a grammar production. If we have multiple nodes in different example trees with the same target, we assume that we can always merge the productions that are generated based on the same target element.

As an example, we can use the following two classes:

```
class AJavaClass {
    public void hello() {
        System.out.println("Hello World!");
    }
}

class AnotherJavaClass {
    private void anotherHello() {
        System.out.println("Hello private Method!");
    }
}
```

If we look at each class individually, we get the following two grammars based on our mapping examples (only the relevant snippets):

```
Class  ::= "class" <IDENTIFIER> "{" Method* "}" ;
Method ::= "public" "void" <IDENTIFIER> "(" ")" "{" not_known* "}" ;

Class  ::= "class" <IDENTIFIER> "{" Method* "}" ;
Method ::= "private" "void" <IDENTIFIER> "(" ")" "{" not_known* "}" ;
```

In this case our assumptions leads to the following grammar:

```
Class  ::= "class" <IDENTIFIER> "{" Method* "}" ;
Method ::= "public" "void" <IDENTIFIER> "(" ")" "{" not_known* "}"
         | "private" "void" <IDENTIFIER> "(" ")" "{" not_known* "}" ;
```

We remove one production for the nonterminal *Class* because both productions are equal. We merge both productions for *Method* based on our assumption.

There are a few cases where this assumption is wrong. This means that we produce a parser that can parse source code that is not valid. For a detailed description of that problem refer to Section 5.2.3.

3.7 Parser Generation

In Section 3.1 we talked about the scanner definition and in Section 3.6 we showed how we build our grammars. We now put these two things together and use SmaCC to generate a parser.

We use the scanner definition without any modifications because we used the SmaCC syntax. We have to export our grammar in a way that SmaCC can work with it. We just write out our grammar rules as a string and give it to SmaCC. SmaCC performs a syntax check of our grammar and gives back useful debug information and error codes. We ignore some of the messages (for example we do not care if there are unused nonterminals in our grammar rules) and some we give to the user. If SmaCC cannot generate a functional parser, we have to review our examples, rebuild a grammar and try to generate a parser again. If SmaCC fails to build a parser, in the majority of cases the problem is that the grammar is ambiguous. We discuss this problem in Section 5.2.1.

3.8 Parse Tree

If we successfully go through all the previous steps we obtain a running parser. Now, we look at the parse tree that we get when using this parser with some input. We have one important reduce action in our parser: The *reduce* action that creates a new *Node* in our parse tree. All other actions just return the source code. Every time a production that is based on a target element matches the actual input we create a new *Node*. This new *Node* knows the following things:

- It knows to which element in the target model it should be mapped.
- It knows its name.
- It knows its parent and its children.
- It knows based on which production in the grammar it is created.
- It knows its source code.

If we compare this list of properties with the list of properties from our example nodes in Section 3.3 we detect similarities, because the nodes we use for our examples and the nodes we use in our parse tree are instances of the same class.

This means that we get a closed loop:
We specify examples, generate a grammar based on them, use SmaCC for creating a parser, parse a system with that parser and receive a structure that is built up exactly in the same way as we built up our examples.

3.9 Model Export

We can export the parse tree that we get as a model. Every *Node* can export itself based on its attached target element. We just traverse the whole tree and export every single node. The target elements for us are *FAMIXEntities*. Every *Node* can be exported as a *FAMIXEntity* that can be added to a MOOSE model.

Because the parse tree and the tree of examples is built with the same nodes it is also possible to export a tree of examples directly as a model without parsing anything. This can be quite useful. If we just want to look at the structure of some simple things we do not have to generate a parser. We can just specify an example and export that.

3.10 External Representation of the Parser Definition

In the reference implementation of this approach we include some functionality to store and load all definitions that we need to rebuild our parsers. We store our definitions in an eXtensible Markup Language (XML) file according to the Document Type Definition (DTD) that we can see in Appendix B.1.

It is also possible to build parsers based on external definitions if the following requirements are fulfilled:

- The definition must be stored as an XML file that is a valid instance of our DTD.
- The scanner definition and grammar must be parsable by SmaCC.
- The reduce actions of the productions in the grammars must be valid method calls to the class *CodeSnooperParser*.

Chapter 4

Validation

MOOSE is an environment for reengineering object-oriented systems that is developed and maintained by the Software Composition Group of the University of Berne [DUCA 05]. MOOSE works on FAMIX models. FAMIX is a meta-model for a language independent representation of object-oriented source code [DEME 01, TICH 00]. MOOSE has the same problem as almost every reengineering tool: We can only use it for systems that are written in a programming language for which we have a parser that can translate the system into a FAMIX model. We can work without problems with systems written in Cincom Smalltalk. There are also parsers available for C/C++ and Java.

To validate our approach we implemented a tool called *CodeSnooper*. *CodeSnooper* supports currently only the FAMIX model as target model. We have some well defined interfaces and it is possible to extend our application with other target models. Figure 4.1 shows the main view of *CodeSnooper* and the way we specify examples: We highlight a section in the source code and assign a target to this selection. We provide more information about *CodeSnooper* in Appendix A.

We made two case studies for the validation of our approach. As a first example we took a part of JBoss[1]. This application is written in Java and distributed under an open source license. It is an implementation of the J2EE[2] (Java 2 Platform, Enterprise Edition) application server specifications. In the second case study we worked with the main libraries that are distributed with Ruby[3]. Ruby is an object-oriented scripting language.

The procedure is the same for both of the case studies. We take the source code and make three iterations with the following goals:

1. In the first iteration we want to get a quick overview of the system as quickly as possible. We want to detect *Classes* in the system. In the Ruby case study we also detect *Namespaces* in this iteration.
2. In the second iteration we want to add *Methods* to the *Classes*.
3. Beside *Classes* and *Methods* we also add *Attributes* to the model in the third iteration.

For every iteration we provide the following information:

- **Definition of Examples**: What examples do we specify to reach the goal of that iteration? What does the grammar look like? Sometimes we do more than one pass with different examples. We

[1]`http://www.jboss.org/`
[2]`http://java.sun.com/j2ee/`
[3]`http://www.ruby-lang.org/en/`

provide all grammars that we get. In these grammars there are also nonterminals included for which we do not provide a definition. These are the productions which contain the tokens we want to ignore. As an example such a grammar rule can look as follows:

```
common_not_known ::=
        <DECIMAL_INTEGER>
    |   <HEX_INTEGER>
    |   <OCTAL_INTEGER>
    |   <IDENTIFIER>
    |   <comment>
    |   <eol>
    |   <whitespaces>
    |   "[" common_not_known* "]"
    |   "." | "," | ":" | ";" | "+" | "-" | "=" | "'" | "*" | "|"
    |   "/" | "\" | "&" | "%" | "!" | "?" | "<" | ">" | "~" | "^"
    |   "@" | """" ;
```

What symbols are in these rules depends on the mapping examples. For instance, we do not include the semicolon in such an ignore rule if we have an example of an *Attribute* in Java that uses the semicolon for its definition.

- **Results**: What do we get from these examples? First, we count how many files we can parse. If our parser can parse a file completely without errors we say that this is a valid parsed file. That means that we have *Nodes* for that file that we can export to a model. Then we export those *Nodes* to a FAMIX model and examine what we really get. In each case we show the relevant parts of the system overview in MOOSE. Figure A.6 shows the main view of MOOSE where we loaded a model of the jboss-j2ee package. In the end we discuss if we can tune anything to get other results.

- **Problems**: What are the problems in this iteration? Why are there files that our parser cannot parse? We look only in the Java case study at the problems separately for each iteration. In the Ruby case study we summarize the problems at the end.

- **Comparison with a Precise Model**: How does the real world look like? What is not in our model? Why are there any entities that we cannot detect?

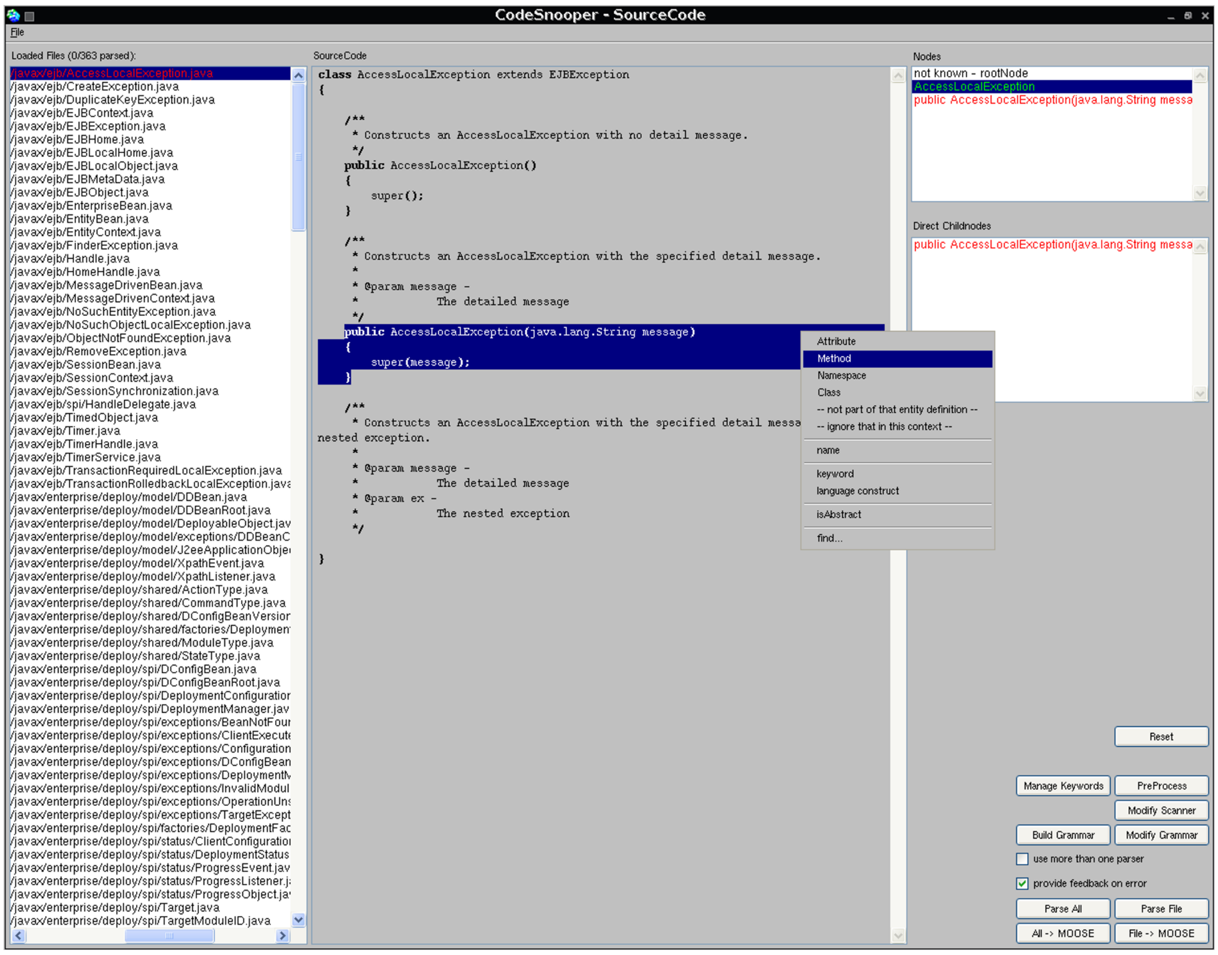
4.1 JBoss j2ee Package

As first case study we worked with an open source implementation of the J2EE application server specifications. We analyzed the jboss-j2ee package of JBoss. In this package and its subpackages are 363 Java files. We checked out the source code directly from the CVS[4] repository (Concurrent Versions System).

We do not change the scanner definition for any of the following iterations. We use the definition as it is written in Section 3.1. We also show some grammar rules. These are only the important parts of them. In Appendix B.3 we show one complete grammar as an example.

Because we care about whitespace characters (see Section 3.6.3) it can be difficult to recognize some elements in source code with an arbitrary formatting. It does not matter how the code is formatted but it should be the same in every file. We use the code formatting facility from eclipse[5] to bring all source code files into the same shape.

[4]https://www.cvshome.org/
[5]http://www.eclipse.org/



Loaded Files (0/363 parsed):

```
/javax/ejb/AccessLocalException.java
/javax/ejb/CreateException.java
/javax/ejb/DuplicateKeyException.java
/javax/ejb/EJBContext.java
/javax/ejb/EJBException.java
/javax/ejb/EJBHome.java
/javax/ejb/EJBLocalHome.java
/javax/ejb/EJBLocalObject.java
/javax/ejb/EJBMetaData.java
/javax/ejb/EJBObject.java
/javax/ejb/EnterpriseBean.java
/javax/ejb/EntityBean.java
/javax/ejb/EntityContext.java
/javax/ejb/FinderException.java
/javax/ejb/Handle.java
/javax/ejb/HomeHandle.java
/javax/ejb/MessageDrivenBean.java
/javax/ejb/MessageDrivenContext.java
/javax/ejb/NoSuchEntityException.java
/javax/ejb/NoSuchObjectLocalException.java
/javax/ejb/ObjectNotFoundException.java
/javax/ejb/RemoveException.java
/javax/ejb/SessionBean.java
/javax/ejb/SessionContext.java
/javax/ejb/SessionSynchronization.java
/javax/ejb/spi/HandleDelegate.java
/javax/ejb/TimedObject.java
/javax/ejb/Timer.java
/javax/ejb/TimerHandle.java
/javax/ejb/TimerService.java
/javax/ejb/TransactionRequiredLocalException.java
/javax/ejb/TransactionRolledbackLocalException.java
/javax/enterprise/deploy/model/DDBean.java
/javax/enterprise/deploy/model/DDBeanRoot.java
/javax/enterprise/deploy/model/DeployableObject.jav
/javax/enterprise/deploy/model/exceptions/DDBeanC
/javax/enterprise/deploy/model/J2eeApplicationObje
/javax/enterprise/deploy/model/XpathEvent.java
/javax/enterprise/deploy/model/XpathListener.java
/javax/enterprise/deploy/shared/ActionType.java
/javax/enterprise/deploy/shared/CommandType.java
/javax/enterprise/deploy/shared/DConfigBeanVersior
/javax/enterprise/deploy/shared/factories/Deploymen
/javax/enterprise/deploy/shared/ModuleType.java
/javax/enterprise/deploy/shared/StateType.java
/javax/enterprise/deploy/spi/DConfigBean.java
/javax/enterprise/deploy/spi/DConfigBeanRoot.java
/javax/enterprise/deploy/spi/DeploymentConfiguratior
/javax/enterprise/deploy/spi/DeploymentManager.jav
/javax/enterprise/deploy/spi/exceptions/BeanNotFour
/javax/enterprise/deploy/spi/exceptions/ClientExecute
/javax/enterprise/deploy/spi/exceptions/Configuration
/javax/enterprise/deploy/spi/exceptions/DConfigBean
/javax/enterprise/deploy/spi/exceptions/DeploymentM
/javax/enterprise/deploy/spi/exceptions/InvalidModul
/javax/enterprise/deploy/spi/exceptions/OperationUns
/javax/enterprise/deploy/spi/exceptions/TargetExcept
/javax/enterprise/deploy/spi/factories/DeploymentFac
/javax/enterprise/deploy/spi/status/ClientConfiguratio
/javax/enterprise/deploy/spi/status/DeploymentStatus
/javax/enterprise/deploy/spi/status/ProgressEvent.jav
/javax/enterprise/deploy/spi/status/ProgressListener.j
/javax/enterprise/deploy/spi/status/ProgressObject.ja
/javax/enterprise/deploy/spi/Target.java
/javax/enterprise/deploy/spi/TargetModuleID.java
```

SourceCode

```
class AccessLocalException extends EJBException
{

    /**
     * Constructs an AccessLocalException with no detail message.
     */
    public AccessLocalException()
    {
        super();
    }

    /**
     * Constructs an AccessLocalException with the specified detail message.
     *
     * @param message -
     *            The detailed message
     */
    public AccessLocalException(java.lang.String message)
    {
        super(message);
    }

    /**
     * Constructs an AccessLocalException with the specified detail messa
nested exception.
     *
     * @param message -
     *            The detailed message
     * @param ex -
     *            The nested exception
     */

}
```

Nodes

```
not known - rootNode
AccessLocalException
public AccessLocalException(java.lang.String messa
```

Direct Childnodes

```
public AccessLocalException(java.lang.String messa
```

Figure 4.1: CodeSnooper: Main view while specifying an example.

4.1.1 First Iteration: Detection of Classes

Definition of Examples

We specify three examples in three different files:

- The first example is a normal Java class that we map to a FAMIXClass without any attributes set.
  `javax/ejb/AccessLocalException.java`
- The second example is an abstract Java class that we map to a FAMIXClass. For that entity we also set the 'isAbstract' attribute to true.
  `javax/xml/soap/SOAPPart.java`
- As third example we take a Java interface that we map also to a FAMIXClass. We also set the 'isAbstract' attribute for that entity to true.
  `javax/ejb/EJBObject.java`

We get the following grammar productions from these examples (without reduce actions):

```
Class ::=  "class" <IDENTIFIER> 'name' not_known* "{" Class_not_known* "}"
      | "interface" <IDENTIFIER> 'name' not_known* "{" Class_not_known* "}"
      | "abstract" "class" <IDENTIFIER> not_known* "{" Class_not_known* "}" ;
```

We say that we want to ignore the keyword "**abstract**" in the context of a class. Otherwise our parser would fail whenever it finds an abstract method.

Results

We try to parse every file with that parser. We can parse 355 of the 363 Java files at first go. In Table 4.1 we see the system overview measurements of MOOSE after the import of our model.

Number of Model Classes	355
Number of Abstract Classes	229
Total Number Of Methods	0
Total Number of Attributes	0

Table 4.1: System overview in MOOSE after first import (Java - first iteration).

Problems

There are eight files that we cannot parse with our grammar. In these file we can find for example the following code snippets:

```
new PolicyContextException("Failure during load of class: "+action.getName(), e);

System.getProperty(SOAPFactory.class.getName(), DEFAULT\_FACTORY);

private static class MatchingIter implements Iterator
```

In all of these examples there is somewhere the keyword “**class**” contained. In the first case it is in a String, in the second case there it is used to access a *Class* object and in the third case we can see the definition of an inner class. These occurrences of “**class**” break the parsing of these files because this keyword is not expected to be there. We want to parse those files anyway and decide to ignore the keyword “**class**” in the context of a *Class*. So, we can parse 361 Java files. We see the system overview measurements after exporting our model to MOOSE in Table 4.2.

Number of Model Classes	361
Number of Abstract Classes	233
Total Number Of Methods	0
Total Number of Attributes	0

Table 4.2: System overview in MOOSE after second import (Java - first iteration).

In the remaining two files we have the following problems:

- **`javax/security/jacc/EJBMethodPermission.java`**: In this file we find the following statement:

  ```
  tmp.append('[');
  ```

  With our definitions of language constructs we expect that nested brackets are well formed. In this case here is a squared bracket open as a character and not as a language construct. We cannot make a difference between these two things.

- **`javax/security/jacc/URLPattern.java`**: Here we have the following line of code:

  ```
  else if( pattern.startsWith("/*") )
  ```

  Here starts a multi line comment in a String. Thus, our scanner searches for the end of that comment in order to generate a token for the parser. It just skips source code from here until the next end of a multi line comment: Information get lost. For example the two closing brackets at the end of the line. The rest of the source code is not parsable anymore.

The problems we have in these two files have the same cause. This is described in more detail in Section 5.2.4.

Comparison with a Precise Model

	Precise Model	Our Model
Number of Model Classes	366	361
Number of Abstract Classes	233	233

Table 4.3: Comparison between a precise model and our model (Java - first iteration).

As we see in Table 4.3 we can detect all but five *Model Classes*. Our model includes all of the 233 *Abstract Classes*.

Because we ignored the keyword “**class**” in the body of *Classes* we cannot detect *Inner Classes*. In the whole jboss-j2ee package there are three *Inner Classes*. If we count all our misses (three *Inner Classes* and two files that we cannot parse) we have five *Classes* that we cannot detect. Our model contains almost **99 percent** of all target elements for this iteration.

Because we do not detect packages we must take care that we do not map two different *Classes* on the same entity in our model. This problem and our solution is described in Section 5.2.6.

4.1.2 Second Iteration: Detection of Classes and Methods

Definition of Examples

We make three passes. First, we work with an *Abstract Class*, then with a concrete *Class* and after that we look at an interface that we map to an *Abstract Class*. In the end, we put it all together.

First Pass: Abstract Class
A problem is the definition of an abstract *Method* because it is quite different than that of a concrete *Method*. In this pass we take an abstract *Class* and specify an example of an abstract *Method* in this file:
`javax/xml/soap/SOAPConnection.java`
That example gives us the following grammar:

```
Class           ::= "abstract" "class" <IDENTIFIER> 'name' not_known*
                    "{" ( Class_not_known | Method )* "}"                ;
Method          ::= "abstract" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
                    "(" not_known* ")" not_known* ";"                    ;
Class_not_known ::= common_not_known | "(" Class_not_known* ")" | ";"    ;
```

Second Pass: Concrete Class
We do the same as for the *Abstract Class* before for a concrete *Class*. We choose the following file:
`java/jms/TopicRequestor.java`
In this file we make an example for the *Class* itself and two of its *Methods*. One of them is a constructor. We get the following grammar from these examples:

```
Class           ::= "class" <IDENTIFIER> 'name' not_known*
                    "{" ( Class_not_known | Method )* "}"                   ;
Method          ::= "public" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name' "("
                    not_known* ")" not_known* "{" Method_not_known* "}"
                  | "public" <IDENTIFIER> 'name' "("  not_known* ")"
                    not_known* "{" Method_not_known* "}"                    ;
Class_not_known ::= common_not_known | "(" Class_not_known* ")"             ;
```

As we can see in the definitions of a *Method* we specified "**public**" as a keyword. So, we run into problems because we do not detect *Attributes* until now. If there is an *Attribute* with the visibility "**public**" parsing fails. With this grammar we cannot detect any *Method* with the visibility "**private**" or "**protected**". If we give an example of a private *Method* we can hardly parse any file because our parser breaks whenever it finds a private *Attribute*.

Third Pass: Interface
We take the same interface as in the first iteration and specify one (abstract) *Method* in the already defined (abstract) *Class*. This leads to the following grammar:

```
Class           ::= "interface" <IDENTIFIER> 'name' not_known*
                    "{" ( Class_not_known | Method )* "}"                   ;
Method          ::=  <IDENTIFIER> 'name' "(" not_known* ")" not_known* ";" ;
Class_not_known ::= common_not_known | ";"                                  ;
```

As we see in this grammar we ignore the semicolon in the context of a *Class*. Otherwise we cannot parse any file that contains an *Attribute* definition. Such a definition contains a semicolon, too. Our parser complains about that token if we do not ignore it. Nevertheless we detect *Methods* based on the brackets together with a semicolon.

Results

First Pass: Abstract Class
When we parse all files with the parser generated based on the grammar for this pass we get the results shown in Table 4.4. We recognize only abstract *Methods* with this parser. We cannot detect concrete *Methods* that are defined in those abstract *Classes*.

Number of Model Classes	12
Number of Abstract Classes	12
Total Number Of Methods	64
Total Number of Attributes	0

Table 4.4: System overview in MOOSE after import of abstract classes and methods (Java - second iteration).

Second Pass: Concrete Class
In Table 4.5 we see the results from this run.

Number of Model Classes	83
Number of Abstract Classes	0
Total Number Of Methods	295
Total Number of Attributes	0

Table 4.5: System overview in MOOSE after import of classes and methods (Java - second iteration).

Third Pass: Interface
With the parser based on the grammar for the third pass we detect all of the 221 interfaces defined in the jboss-j2ee package. In Table 4.6 we see the system overview after the import of that model in MOOSE.

Number of Model Classes	221
Number of Abstract Classes	221
Total Number Of Methods	1289
Total Number of Attributes	0

Table 4.6: System overview in MOOSE after import of interfaces (Java - second iteration).

All Three Passes Together
We cannot merge the grammars of the three passes together because of two things:

- The grammar becomes ambiguous. That means we cannot produce a useful parser anymore. We discuss this problem in Section 5.2.1.
- We could detect wrong elements with the merged grammar. Section 5.2.3 contains an explication of that problem.

Nevertheless we can use all of the three parsers in parallel. The result of that run is shown in Table 4.7.

Number of Model Classes	316
Number of Abstract Classes	233
Total Number Of Methods	1648
Total Number of Attributes	0

Table 4.7: System overview in MOOSE after import (Java - second iteration).

Problems

There are still the same two files as in the first iteration that we cannot parse. So, we cannot detect anything in these files. There are the following definitions of *Methods*:

- **`javax/security/jacc/EJBMethodPermission.java`**: This *Class* has a total of ten *Methods*. Three of them are constructors.
- **`javax/security/jacc/URLPattern.java`**: Here are four *Methods* containing one constructor.

There are different kinds of problems in the other files we cannot parse:

- There are definitions of *Attributes* that contain keywords that we use for specifying *Methods*.
- There are *LanguageConstructs* in contexts where we do not expect them. For example one *Class* contains the following source code:

```
static {
    ALL_HTTP_METHODS.add("GET");
    ...
}
```

There is no example that can be matched to that construct. So, our parser does not expect that there can be curly brackets and fails to parse that file. We discuss that problem in Section 5.2.4.

- There are too many different definitions of *Methods* used in one single *Class*. This could be solved by providing more examples.

Comparison with a Precise Model

Table 4.8 shows the comparison between a precise model and our model. As we can see our model includes all of the 233 *Abstract Classes*. This is the combination of the 221 interfaces and the 12 *Abstract Classes*. We can parse 83 of the remaining 133 *Classes* together with their *Methods*. Altogether, we can say that our model at the end of this iteration contains a little more than **85 percent** of all elements we wanted to detect.

	Precise Model	Our Model
Number of Model Classes	366	316
Number of Abstract Classes	233	233
Total Number Of Methods	1887	1648

Table 4.8: Comparison between a precise model and our model (Java - second iteration).

4.1.3 Third Iteration: Detection of Classes, Methods and Attributes

Definition of Examples

We use the same three passes for this iteration as before. First we look only at an *Abstract Class*, then we take a concrete *Class* and after that we work with an interface that we map to an *Abstract Class*. Finally, we look at the combined model of the three passes.

First Pass: Abstract Class
We specify examples in four different files:
`javax/xml/bind/JAXBContext.java`
`javax/xml/rpc/ServiceFactory.java`
`javax/xml/soap/SOAPFactory.java`
`javax/xml/soap/SOAPMessage.java`
We specify an example for an *Abstract Class* in every file. We give totally five examples of *Attributes* and five examples of *Methods*. Two of the *Attribute* definitions are equal and thus filtered out while generating the grammar.

```
Class     ::= "abstract" "class" <IDENTIFIER> 'name' not_known*
              "{" ( Class_not_known | Method | Attribute )* "}"           ;
Method    ::= "public" "static" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
              "(" not_known* ")" not_known* "{" Method_not_known* "}"
          |   "public" <IDENTIFIER> 'name' "(" ")" not_known*
              "{" not_known* "}"
          |   "public" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
              "(" not_known* ")" not_known* "{" Method_not_known* "}"
          |   "public" "abstract" <IDENTIFIER> ("[" "]")? <IDENTIFIER>
              'name' "(" not_known* ")" not_known* ";"
          |   "protected" <IDENTIFIER> 'name' "(" ")" not_known*
              "{" not_known* "}"                                          ;
Attribute ::= "public" "static" "final" <IDENTIFIER> ("[" "]")?
              <IDENTIFIER> 'name' not_known* ";"
```

```
           |   "private" "static" <IDENTIFIER> ("[" "]")? <IDENTIFIER>
               'name' ";"
           |   "private" "static" "final" <IDENTIFIER> ("[" "]")?
               <IDENTIFIER> 'name' not_known* ";"
           |   "private" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
               Attribute_not_known* ";"                                 ;
```

Second Pass: Concrete Class
We choose the same file as in the previous iteration and specify an *Attribute* beside the *Class* and the two *Methods*. We generate the following grammar from this example:

```
Class      ::= "class" <IDENTIFIER> 'name' not_known*
               "{" ( Class_not_known | Method | Attribute )* "}"        ;
Method     ::= "public" <IDENTIFIER> 'name' "(" not_known* ")" not_known*
               "{"  Method_not_known* "}"
           |   "public" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
               "(" not_known* ")" not_known* "{" Method_not_known* "}"  ;
Attribute ::= "private" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
               not_known* ";"                                           ;
```

We have “**public**” and “**private**” as keywords in this grammar. Because we have only private *Attributes* in this *Class* we make some more examples in different files.

As second file we choose:
`javax/xml/rpc/ParameterMode.java`
Here we specify besides the *Class* a *Method* (a private constructor) and an *Attribute* that has class scope. We can safely merge the grammar from the previous example with the grammar that we get from this example. The composed grammar looks as shown here:

```
Class      ::= "class" <IDENTIFIER> 'name' not_known*
               "{" ( Class_not_known | Method | Attribute )* "}"        ;
Method     ::= "public" <IDENTIFIER> 'name' "(" not_known* ")" not_known*
               "{" Method_not_known* "}"
           |   "public" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
               "(" not_known* ")" not_known* "{" Method_not_known* "}"
           |   "private" <IDENTIFIER> 'name' "(" not_known* ")" not_known*
               "{" Method_not_known* "}"                               ;
Attribute ::= "private" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
               not_known* ";"
           |   "public" "static" "final" <IDENTIFIER> ("[" "]")?
               <IDENTIFIER> 'name' Attribute_not_known* ";"             ;
```

We specify examples in three more files:
`javax/enterprise/deploy/shared/CommandType.java`
`javax/resource/spi/ConnectionEvent.java`
`javax/xml/rpc/holders/IntHolder.java`
In these files we find some different kinds of definitions for *Attributes* and *Methods*. In the end we get the following grammar:

```
Class      ::= "class" <IDENTIFIER> 'name' not_known*
               "{" ( Class_not_known | Method | Attribute )* "}"        ;
Method     ::= "public" <IDENTIFIER> 'name' "(" not_known* ")" not_known*
               "{" Method_not_known* "}"
           |   "public" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
               "(" not_known* ")" not_known* "{" Method_not_known* "}"
           |   "public" "static" <IDENTIFIER> ("[" "]")?  <IDENTIFIER> 'name'
               "(" not_known* ")" not_known* "{" Method_not_known* "}"
           |   "protected" <IDENTIFIER> 'name' "(" not_known* ")" not_known*
               "{" Method_not_known* "}"
           |   "protected" <IDENTIFIER> ("[" "]")?  <IDENTIFIER> 'name'
```

```
                "(" ")" not_known* "{" Method_not_known* "}"
          |     "private" <IDENTIFIER> 'name' "(" not_known* ")" not_known*
                "{" Method_not_known* "}"                                   ;
Attribute ::= "public" "static" "final" <IDENTIFIER> ("[" "]")?
                <IDENTIFIER> 'name' Attribute_not_known* ";"
          |     "public" <IDENTIFIER> ("[" "]")?  <IDENTIFIER> 'name' ";"
          |     "protected" <IDENTIFIER> ("[" "]")?  <IDENTIFIER> 'name' ";"
          |     "private" "static" "final" <IDENTIFIER> ("[" "]")?
                <IDENTIFIER> 'name' Attribute_not_known* ";"
          |     "private" <IDENTIFIER> ("[" "]")? <IDENTIFIER> 'name'
                not_known* ";"                                              ;
```

Third Pass: Interface
We take another interface as in the previous iterations because the one we used before does not contain any *Attributes*.
`javax/transaction/xa/XAResource.java`
We specify the mapping to an abstract *Class* and an abstract *Method* as before. We add the definition of an *Attribute* which has class scope (static in Java):

```
static final int TMNOFLAGS = 0;
```

In another file there is the following definition of an *Attribute* without the final-keyword:
`javax/xml/bind/Marshaller.java`:

```
static String JAXB_ENCODING = "jaxb.encoding";
```

We take this file as second example and specify here examples of *Class*, *Method* and *Attribute*, too. Because only the definition of an *Attribute* is different than in the first file, we can merge those grammar productions to the following grammar:

```
Class     ::= "interface" <IDENTIFIER> 'name' not_known*
                "{" ( Class_not_known | Method | Attribute )* "}"          ;
Method    ::= <IDENTIFIER> 'name' "(" not_known* ")" not_known* ";"        ;
Attribute ::= "static" "final" <IDENTIFIER> <IDENTIFIER> 'name'
                Attribute_not_known* ";"
          |     "static" <IDENTIFIER> <IDENTIFIER> 'name'
                Attribute_not_known* ";"                                    ;
```

Results

First Pass: Abstract Class
The parser that is based on the examples of *Abstract Classes* produces the results in Table 4.9. We notice that we detect only ten *Abstract Classes* compared to the twelve from the second iteration. Both of the missing *Classes* contain an *Inner Class*. Its definition is composed of the same keywords as the one for *Methods* and *Attributes*. That is why our parser fails.

Number of Model Classes	10
Number of Abstract Classes	10
Total Number Of Methods	91
Total Number of Attributes	12

Table 4.9: System overview in MOOSE after import of abstract classes, methods and attributes (Java - third iteration).

Second Pass: Concrete Class
With the grammar that we get from the first example we can parse 79 files. With the grammar based on two examples we can parse 84 files and we notice that we detect much more *Attributes* with this grammar. In the end we work with the parser that is based on the examples that we make in five different files. With this parser we can parse 116 files and detect a respectable number of entities. The result of these three passes is shown in Table 4.10.

	Grammar 1	Grammar 2	Grammar 3
Number of Model Classes	79	84	116
Number of Abstract Classes	0	0	0
Total Number Of Methods	267	287	400
Total Number of Attributes	26	73	172

Table 4.10: System overview in MOOSE after import of classes, methods and attributes (Java - third iteration).

Third Pass: Interface
We have examples in two different files here. When we use a parser that is based only on the first example we can parse 219 files after all. With the additional example in another file (Marshaller.java) we come up to 220 files that we can parse. That means that the second example is only used for parsing the file Marshaller.java. But in this file are four *Attributes* and ten *Methods* so it is worth the effort of one more example. In Table 4.11 we can see the system overview after the import of that model in MOOSE. Figure 4.2 shows the main view of *CodeSnooper* after parsing our input with the parser of this pass.

Number of Model Classes	220
Number of Abstract Classes	220
Total Number Of Methods	1289
Total Number of Attributes	120

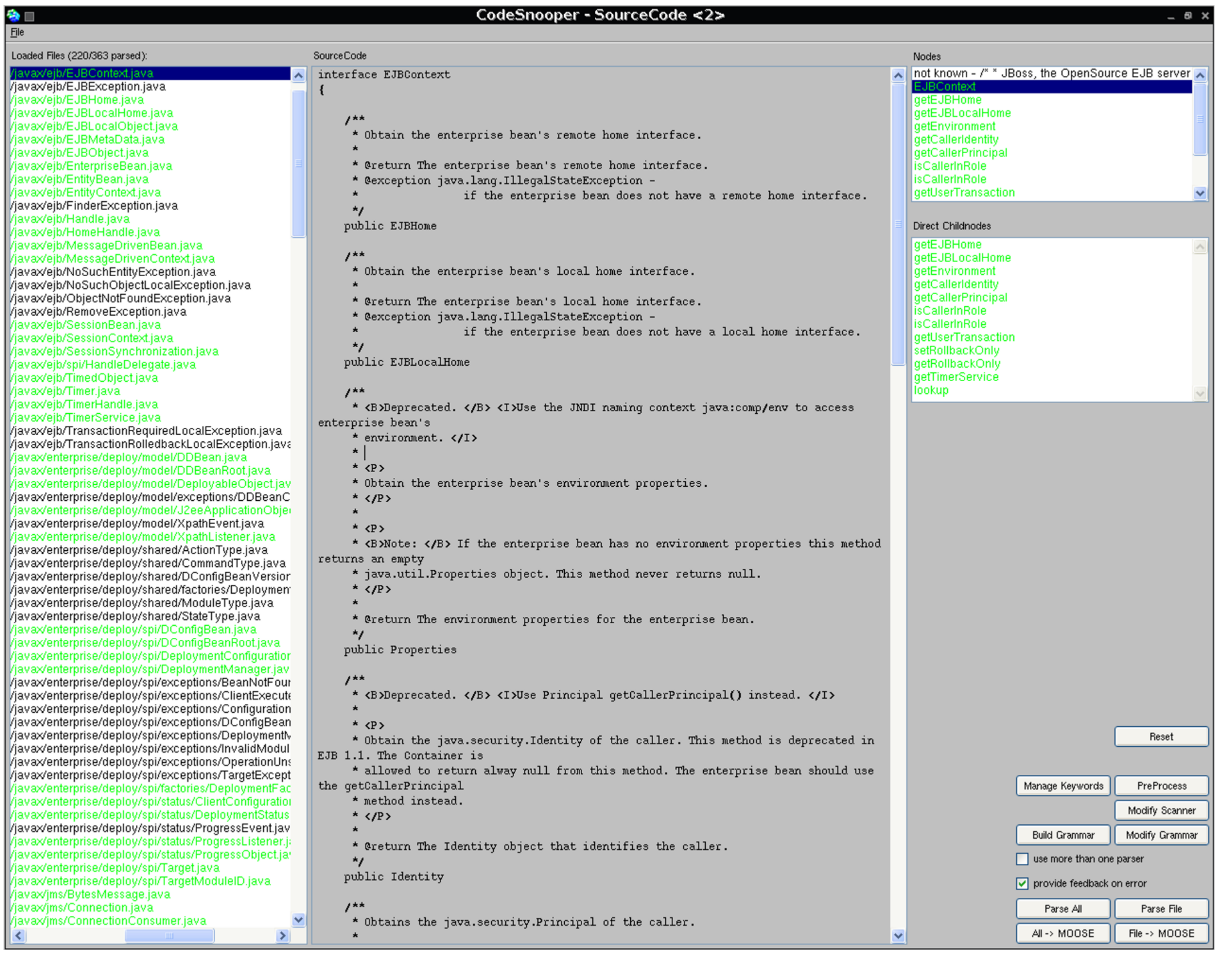
Table 4.11: System overview in MOOSE after import of interfaces (Java - third iteration).

There is one interface that we cannot parse:
`javax/xml/soap/SOAPConstants.java`
In this file we find the following source code:

```
static final String URI_NS_SOAP_ENCODING="http://schemas.xmlsoap.org/soap/encoding/";
```

Here we have a problem with the double slash. This is the sign for a single line comment. So, our scanner skips the rest of the line after the double slash. The rest of the file is then unreadable for our parser. We refer to Section 5.2.4 for a detailed description of that problem. If we compare the result from this pass with the one from the second iteration we notice that we still detect the same number of *Methods*. That means that there are no *Methods* in that file (SOAPConstants.java). In fact there are three *Attributes* defined and no *Methods*.

All Three Passes Together
In this iteration we cannot use the parsers from the three passes in parallel as in the second iteration. In each pass we specified examples in different files. Especially the grammar we use in the second pass gains its power because of the merge. However we can use each of the parsers individually and add all detected entities to our model one after another. The overview of the resulting model is shown in Table 4.12.



```
/javax/ejb/EJBContext.java
/javax/ejb/EJBException.java
/javax/ejb/EJBHome.java
/javax/ejb/EJBLocalHome.java
/javax/ejb/EJBLocalObject.java
/javax/ejb/EJBMetaData.java
/javax/ejb/EJBObject.java
/javax/ejb/EnterpriseBean.java
/javax/ejb/EntityBean.java
/javax/ejb/EntityContext.java
/javax/ejb/FinderException.java
/javax/ejb/Handle.java
/javax/ejb/HomeHandle.java
/javax/ejb/MessageDrivenBean.java
/javax/ejb/MessageDrivenContext.java
/javax/ejb/NoSuchEntityException.java
/javax/ejb/NoSuchObjectLocalException.java
/javax/ejb/ObjectNotFoundException.java
/javax/ejb/RemoveException.java
/javax/ejb/SessionBean.java
/javax/ejb/SessionContext.java
/javax/ejb/SessionSynchronization.java
/javax/ejb/spi/HandleDelegate.java
/javax/ejb/TimedObject.java
/javax/ejb/Timer.java
/javax/ejb/TimerHandle.java
/javax/ejb/TimerService.java
/javax/ejb/TransactionRequiredLocalException.java
/javax/ejb/TransactionRolledbackLocalException.java
/javax/enterprise/deploy/model/DDBean.java
/javax/enterprise/deploy/model/DDBeanRoot.java
/javax/enterprise/deploy/model/DeployableObject.jav
/javax/enterprise/deploy/model/exceptions/DDBeanC
/javax/enterprise/deploy/model/J2eeApplicationObje
/javax/enterprise/deploy/model/XpathEvent.java
/javax/enterprise/deploy/model/XpathListener.java
/javax/enterprise/deploy/shared/ActionType.java
/javax/enterprise/deploy/shared/CommandType.java
/javax/enterprise/deploy/shared/DConfigBeanVersior
/javax/enterprise/deploy/shared/factories/Deploymen
/javax/enterprise/deploy/shared/ModuleType.java
/javax/enterprise/deploy/shared/StateType.java
/javax/enterprise/deploy/spi/DConfigBean.java
/javax/enterprise/deploy/spi/DConfigBeanRoot.java
/javax/enterprise/deploy/spi/DeploymentConfiguratior
/javax/enterprise/deploy/spi/DeploymentManager.jav
/javax/enterprise/deploy/spi/exceptions/BeanNotFour
/javax/enterprise/deploy/spi/exceptions/ClientExecute
/javax/enterprise/deploy/spi/exceptions/Configuration
/javax/enterprise/deploy/spi/exceptions/DConfigBean
/javax/enterprise/deploy/spi/exceptions/DeploymentM
/javax/enterprise/deploy/spi/exceptions/InvalidModul
/javax/enterprise/deploy/spi/exceptions/OperationUns
/javax/enterprise/deploy/spi/exceptions/TargetExcept
/javax/enterprise/deploy/spi/factories/DeploymentFac
/javax/enterprise/deploy/spi/status/ClientConfiguratior
/javax/enterprise/deploy/spi/status/DeploymentStatus
/javax/enterprise/deploy/spi/status/ProgressEvent.jav
/javax/enterprise/deploy/spi/status/ProgressListener.j
/javax/enterprise/deploy/spi/status/ProgressObject.ja
/javax/enterprise/deploy/spi/Target.java
/javax/enterprise/deploy/spi/TargetModuleID.java
/javax/jms/BytesMessage.java
/javax/jms/Connection.java
/javax/jms/ConnectionConsumer.java
```

```
interface EJBContext
{

    /**
     * Obtain the enterprise bean's remote home interface.
     *
     * @return The enterprise bean's remote home interface.
     * @exception java.lang.IllegalStateException -
     *               if the enterprise bean does not have a remote home interface.
     */
    public EJBHome

    /**
     * Obtain the enterprise bean's local home interface.
     *
     * @return The enterprise bean's local home interface.
     * @exception java.lang.IllegalStateException -
     *               if the enterprise bean does not have a local home interface.
     */
    public EJBLocalHome

    /**
     * <B>Deprecated. </B> <I>Use the JNDI naming context java:comp/env to access
enterprise bean's
     * environment. </I>
     *
     * <P>
     * Obtain the enterprise bean's environment properties.
     * </P>
     *
     * <P>
     * <B>Note: </B> If the enterprise bean has no environment properties this method
returns an empty
     * java.util.Properties object. This method never returns null.
     * </P>
     *
     * @return The environment properties for the enterprise bean.
     */
    public Properties

    /**
     * <B>Deprecated. </B> <I>Use Principal getCallerPrincipal() instead. </I>
     *
     * <P>
     * Obtain the java.security.Identity of the caller. This method is deprecated in
EJB 1.1. The Container is
     * allowed to return alway null from this method. The enterprise bean should use
the getCallerPrincipal
     * method instead.
     * </P>
     *
     * @return The Identity object that identifies the caller.
     */
    public Identity

    /**
     * Obtains the java.security.Principal of the caller.
     *
```

```
not known - /* * JBoss, the OpenSource EJB server
EJBContext
getEJBHome
getEJBLocalHome
getEnvironment
getCallerIdentity
getCallerPrincipal
isCallerInRole
isCallerInRole
getUserTransaction
```

```
getEJBHome
getEJBLocalHome
getEnvironment
getCallerIdentity
getCallerPrincipal
isCallerInRole
isCallerInRole
getUserTransaction
setRollbackOnly
getRollbackOnly
getTimerService
lookup
```

Figure 4.2: CodeSnooper: Main view after parsing (Java - third iteration - third pass).

If we compare the result from Table 4.12 to the one of the second iteration (Table 4.7) we see that we detect 30 *Classes* and 132 *Methods* more in this iteration. But we also recognize three *Abstract Classes* less than before.

Number of Model Classes	346
Number of Abstract Classes	230
Total Number Of Methods	1780
Total Number of Attributes	304

Table 4.12: System overview in MOOSE after import (Java - third iteration).

Problems

We have still the same two files as in the first iteration that we cannot parse. In these files are the following *Attribute* definitions:

- **`javax/security/jacc/EJBMethodPermission.java`**: Here we have four *Attributes*.
- **`javax/security/jacc/URLPattern.java`**: In this *Class* are nine *Attributes*. Five of them have class scope.

In the following list we collect some of the problems that we have in the other *Classes* that we cannot parse:

- There are *Attributes* with a double slash in their definition and our parser takes this as a comment. So we cannot detect *Attributes* that are defined in such *Classes*. This is the same problem as in the first pass. We refer to Section 5.2.4 for a discussion of that problem.
- There are even more definitions of *Attributes* and *Methods* in the source code and we do not get all of them.
- In some files there are definitions of *Attributes* and *Methods* without a visibility identifier. This means that their definition does not contain any of the keywords “**public**”, “**protected**” or “**private**”. Because we use these keywords in our grammar for identifying our target elements we do not get the ones without a visibility identifier.

Comparison with a Precise Model

Table 4.13 shows the real number of entities in the jboss-j2ee package and also the number of entities that we detect. We detect almost **95 percent** of all *Classes* and *Methods* from the jboss-j2ee package. We get about **75 percent** of all *Attributes*. There are *Classes* that act as providers for constants and are just containers for *Attributes*. Altogether we can say that we get about **90 percent** of the elements we wanted to detect.

We just provide the numbers of entities in the results in this work. In fact all those entities do also have some properties set. The properties that we currently set include all boolean attributes of the FAMIX model. For example a *Method* can be abstract or can be a constructor.

	Precise Model	Our Model
Number of Model Classes	366	346
Number of Abstract Classes	233	230
Total Number Of Methods	1887	1780
Total Number of Attributes	395	304

Table 4.13: Comparison between a precise model and our model (Java - third iteration).

4.1.4 Comparison with another Parser

We also generate a model from the jboss-j2ee package for MOOSE with the jFamix[6] plugin for eclipse. With jFamix we generate an XMI file that we can load into MOOSE. In Table 4.14 we can see the main data of this model. JFamix takes its information from the model that eclipse itself has. This is a precise parser and what we get is a perfect mapping to our model. We also have six *Library Classes* in this model which do not belong to the jboss-j2ee package. With *CodeSnooper* we do not reach the performance and the precision of a model generator like jFamix. But we see that we detect a lot of things in the source code with our approach without the need to learn anything about parsing. At the moment we do not include the number of lines of code and the number of statements in the model generated with *CodeSnooper*.

[6] `http://jfamix.sourceforge.net/`

Number of Classes	372
Number of Model Classes	366
Number of Library Classes	6
Number of Abstract Classes	233
Total Weighted Number of Lines of Code	5847
Total Number Of Methods	1887
Total Number of Attributes	395
Total Weighted Number of Statements	1039

Table 4.14: jboss-j2ee package loaded into MOOSE after model export with jFamix.

4.2 Ruby Libraries

As second case study we take the libraries distributed with Ruby. We use Ruby version 1.8.2 that was released at the end of 2004. The libraries contain 479 files of source code written in Ruby. We do not have a precise parser for Ruby that can generate a MOOSE model. During this case study we work with a small subset of the 479 files. We recount the recognized elements by hand in order to validate our work. We summarize the problems that we had while parsing Ruby code in Section 4.2.6.

As a subset of the files we take Ruby's standard unit testing library/software. This part of the library contains 22 files written in Ruby. We find all possible definitons of *Modules*, *Classes*, *Methods* and *Attributes* in these files.

In Ruby there are *Classes* and *Modules*. *Modules* are collections of *Methods* and *Constants*. They cannot generate instances. However they can be mixed into *Classes* and other *Modules*. A *Module* cannot inherit from anything. Beside providing mixin functionality, *Modules* do also have the function of *Namespaces*. Ruby does not support *Abstract Classes* [MATS 02].

For this case study we map *Modules* to *FAMIXNamespaces* and *Classes* to *FAMIXClasses*.

For the definition of the scanner tokens for identifiers and comments we use the following regular expressions:

```
<IDENTIFIER>: [a-zA-Z_$] \w* ( \? | \! )?   ;
<comment>:    \# [^\r\n]* <eol>             ;
```

4.2.1 First Iteration: Detection of Classes

Definition of Examples

We specify examples only in one file:
`test/unit/assertions.rb`
We give two examples of *Namespaces* and also two examples of *Classes*. We need all of them to express the relationships between *Namespaces* and *Classes*. We specify a *Namespace* on top and another one that is contained in the first one. We also specify a *Class* in a *Namespace* and another *Class* that is contained in the first *Class*. We build the following grammar based on these examples:

```
Namespace ::= "module" <IDENTIFIER> 'name' ( not_known | Namespace |
              Class )* "end"                                          ;
Class     ::= "class" <IDENTIFIER> 'name' ( not_known | Class )* "end"     ;
```

Results

With the parser based on that grammar we parse 7 of the 22 files. The system overview after the import of them into MOOSE is shown in Table 4.15.

Number of Namespaces	6
Number of Model Classes	4
Total Number Of Methods	0
Total Number of Attributes	0

Table 4.15: System overview in MOOSE after import (Ruby - first iteration).

Comparison with a Precise Model

In Table 4.16 we see a comparison between the entities we detected and the entities that are in the source code. We also include a column where we give the number of entities that are in those 7 source files that we parsed. We notice that we get all information out of the files that we successfully parsed. We detect **75 percent** of the *Namespaces* and **16 percent** of the *Classes* compared to the source code that we have as input.

	Precise Model	7 files	Our Model
Number of Namespaces	8	6	6
Number of Model Classes	25	4	4

Table 4.16: Comparison between a precise model and our model (Ruby - first iteration).

4.2.2 Second Iteration: Detection of Classes and Methods

Definition of Examples

We take the examples of *Namespaces* and *Classes* from the first iteration and add two definitions of *Methods*. One *Method* in a *Module* and one in a *Class*. So, the grammar is adjusted that it looks like that:

```
Namespace ::= "module" <IDENTIFIER> 'name' ( not_known | Method |
              Namespace | Class )* "end"                          ;
Class     ::= "class" <IDENTIFIER> 'name' ( not_known | Method | Class )*
              "end"                                                ;
Method    ::= "def" <IDENTIFIER> 'name' not_known* "end"           ;
```

Results

In this iteration we also parse 7 of the 22 files. Table 4.17 shows the overview of the results from this iteration. We see that we still detect six *Namespaces* and four *Classes*. New are the six *Library Classes* and 26 *Methods*. In Ruby a *Module* contains *Methods* but in our model a *Namespace* cannot. To not lose such *Methods* we create *Library Classes* and attach the *Methods* to them. Such a *Library Class* has the same name as the *Namespace* to which we would like to add the *Method*. The obtained model is quite a good reflection of the real circumstances.

Number of Namespaces	6
Number of Classes	10
Number of Model Classes	4
Number of Library Classes	6
Total Number Of Methods	26
Total Number of Attributes	0

Table 4.17: System overview in MOOSE after import (Ruby - second iteration).

Comparison with a Precise Model

If we compare our model with precise one we notice that we can detect only around **10 percent** of all *Methods*. Still, we get all information out of the 7 files that we can successfully parse. The numbers are shown in Table 4.18.

	Precise Model	7 files	Our Model
Number of Namespaces	8	6	6
Number of Model Classes	25	4	4
Total Number Of Methods	247	26	26

Table 4.18: Comparison between a precise model and our model (Ruby - second iteration).

4.2.3 Third Iteration: Detection of Classes, Methods and Attributes

Definition of Examples

We do not change our examples from the second iteration. We add two *Attribute* definitions: one in a *Method* and one in a *Class* because in Ruby we must not declare our *Attributes* somewhere outside of *Methods*.

In a first run we just work with the grammar generated based on these examples:

```
Namespace ::= "module" <IDENTIFIER> 'name' ( not_known | Method |
              Namespace | Class )* "end"                              ;
Class     ::= "class" <IDENTIFIER> 'name' ( not_known | Method |
              Attribute | Class )* "end"                              ;
Method    ::= "def" <IDENTIFIER> 'name' ( not_known | Attribute )* "end"   ;
Attribute ::= "@" <IDENTIFIER> 'name'                                 ;
```

In this first run we only parse four files. All problems have the same cause: We find some *Attributes* inside of brackets. Because we ignore all constructs with brackets our parser does not expect an *Attribute* inside of brackets. In this case it is not enough to specify another example because that would conflict with our *ignore productions*. We change the following grammar production to the one below it:

```
not_known ::= "(" not_known* ")"                                      ;

not_known ::= "(" ( not_known | Attribute )* ")"                      ;
```

It is correct to detect an *Attribute* inside these brackets. It is included in our parse tree as all other *Nodes* too. The only difference to a *Node* generated by the original productions is that there is at least one *Node* without a target between this *Attribute-Node* and its real parent (a *Method-Node* or a *Class-Node*).

Results

With the parser based on this slightly modified grammar we can parse the same 7 files as in the iterations before. In Table 4.19 we show what we get from this run. Besides the 9 new *Attributes* everything is the same as before.

Number of Namespaces	6
Number of Classes	10
Number of Model Classes	4
Number of Library Classes	6
Total Number Of Methods	26
Total Number of Attributes	9

Table 4.19: System overview in MOOSE after import (Ruby - third iteration).

Comparison with a Precise Model

Table 4.20 contains an overview of our final model. We have almost the same detection rate for *Attributes* as we have for *Methods*. We recognize around **7 percent** of all *Attributes*. In Section 4.2.5 we discuss why we do not recognize more elements.

	Precise Model	7 files	Our Model
Number of Namespaces	8	6	6
Number of Model Classes	25	4	4
Total Number Of Methods	247	26	26
Total Number Of Attributes	136	9	9

Table 4.20: Comparison between a precise model and our model (Ruby - third iteration).

4.2.4 Comparison with another Parser

Unfortunately there is no other parser available that can transform Ruby source code into a model that is usable by MOOSE. However we use some unix command line tools (find, grep, sed, wc, uniq) for getting an overview of the elements that are in the complete Ruby libraries. We also use the parsers from the three iterations with all of the library files. With all of these tests we get the same results: We can parse around **30 percent** of the files and detect around **15 percent** of all *Classes* and **10 percent** of all *Methods* and *Attributes*. But we always get all of the information from a file that we can parse. In Section 4.2.5 we discuss why our parsers perform so badly in this case study and in Section 4.2.6 we look at some general problems that we have while parsing source code written in Ruby.

4.2.5 Conclusions

In the set of 22 files there are four files that contain graphical user interfaces for the Ruby testing library. These 4 files contain 10 *Classes*, 147 *Methods* and 92 *Attributes*. If we cut these 4 files we detect around **25 percent** of all target elements.

We also investigate why we cannot parse the rest of the files. There are the two main reasons for that:

1. We often find a hash character "#" in a string or in a regular expression. Our scanner skips the rest of the actual line of source code because the hash is also used as start element for comments. We can handle such situations with some clever scanner tokens. Unfortunately, the scanners generated by SmaCC perform badly if we use complex token definitions. We made some tests with more complex tokens and had success with small files. As soon as we increase our input the scanner stops to deliver tokens. We look at the problem with the hash character in more detail in Section 4.2.6.

2. The control structures of Ruby can be very complex and we have to care about them because they use the same keywords for defining almost every structure. Our implementation of such language constructs is too simple for handling all possibilities that Ruby gives to the programmers.

Our experience shows that we can increase the detection rate of our target elements in source code written in Ruby to around **65-85 percent** when we do some preprocessing of the code. With preprocessing we mean at the moment just to cut all hash characters that do not specify a comment or to cut all comments. We increase the rate even more when we work with more complex language constructs. Until now, we do not have a working implementation of such constructs but it is possible to handle them with our approach as discussed in Section 5.2.5.

4.2.6 Summary of Problems with Ruby Source Code

If we compare the results from this case study with the ones from the Java case study it is obvious that it is harder to parse Ruby source code than Java source code. In this section we want to identify problems that arise when parsing source code written in Ruby. We discuss these problems in Chapter 5 in a more general way and look if we can provide a solution for them with our approach.

In the following list we collect the things that make it difficult to handle Ruby source code with our approach:

- **Complex control structures**: Here are some code snippets from the libraries that are distributed with Ruby.

```
return false unless(other.kind_of?(self.class))

return true if(@filters.empty?)

if(@file.directory?(e_name))
  next if /\ACVS\z/ =~ e
  sub_suite = recursive_collect(e_name, already_gathered)
  sub_suites << sub_suite unless(sub_suite.empty?)
else
  next if /~\z/ =~ e_name or /\A\.\#/ =~ e
  if @pattern and !@pattern.empty?
    next unless @pattern.any? {|pat| pat =~ e_name}
  end
  if @exclude and !@exclude.empty?
    next if @exclude.any? {|pat| pat =~ e_name}
  end
  collect_file(e_name, sub_suites, already_gathered)
end
```

  We see in these few lines that there are a lot of possibilities to use control structures. We can combine (return, if, then, elsif, else, unless, do, next, begin, case, when) statements in almost every combination that makes sense. Often the brackets are optional, but sometimes they are needed. It becomes complicated when we start to nest such constructs. We discuss the problem with such complex control structures in Section 5.2.5.

We can also use regular expressions directly in a statement. For example in the code snippet we find a hash character "`#`" in a regular expression. The hash can only be there because it is in a regular expression. Otherwise we can use the hash for starting a comment. This is a problem because we do not expect this symbol in that context. We discuss this problem in Section 5.2.4.

- **Almost all control structures end with the same token**: The keyword "**end**" is used everywhere in Ruby. It is used in the definition of *Modules*, *Classes*, *Methods* and in control structures like *if-elsif-else-end*, *case-when-then-end* or *begin-end*. That means if we want to detect *Classes* for example we are not allowed to ignore the keywords "**class**" and "**end**" because they belong to the definition of a *Class*. So, we must detect all other structures that use "**end**" for its definition as well. Otherwise we cannot guarantee that we find the right "*end*" for a *Class*. This is difficult with Ruby source code because the control structures can be quite complex. This problem is discussed in Section 5.2.5.

- **String expression substitution**: In Ruby `"#{expression}"` substitutes the value of *expression* in a double-quoted string. This is a useful feature of the language, but it is a problem for our parsers. Whenever our scanner finds a hash "`#`" it just skips the rest of that line because a comment starts with a hash. This leads to the same problem that we meet all the time: we have a language construct in a context where we do not expect it. To solve that problem we can work with tricky scanner definitions or start to detect strings (see Section 5.2.4).

- **More than one possibility to do the same thing**: In Ruby there is often more than one way to express something. We give examples here of regular expressions and quotes. As a remark: the "`{}`" in "`%q{}`" or "`%Q{}`" can also be represented with other characters. There are similar ways to express symbols, command strings and arrays. As we can see in the first point of this list it is also possible to build up control structures in the fanciest ways.

```
regular expressions:
/\d+/     %r{\d+}     %r/\d+/     %r@\d+@     %r(\d+)

quotes:
''        %q{}
""        %Q{}        %{}
```

  All those different possibilities help to increase the complexity of the parser we must build in order to parse source code written in Ruby.

- **Debug and error statements in distributed libraries**: In this case study we look at libraries that are distributed with the programming language itself. We find a lot of debug statements and error handling code in these source files. Reflection and introspection is needed in order to analyze the state of objects and also *Classes*. We find many accesses to *Classes* or *Meta-Classes*. Beside that, the programmers of the libraries often work with regular expressions in the code. In such expressions they can use any characters. In such cases we normally have a keyword in a context where we do not expect it. For example the "**class**" keyword when accessing the *Class* of an object or a hash "**#**" in a regular expression. We encounter this problem quite often and describe it in more detail in Section 5.2.4. It may be that we get better results when working with an industrial project if they work with some strict coding conventions for that project.

The goal of Ruby is to make programming easy and enjoyable [MATS 02]. This is for the programmer who uses the language. The drawback is that everyone who uses the language can write code in a really unique style.

We want to get more information about parsing such complex languages as Ruby. If a programmer uses all possibilities that he has when using Ruby the resulting source code becomes difficult to read even for a human. We do not even know one editor or one integrated development environment (IDE) that can make a totally correct code highlighting of Ruby source code. But there must be available at least one precise parser for Ruby. Namely the one that parses the source code before the interpreter can use it. The definition of this parser is even included in the Ruby distribution. It is there as Yacc source - so, it is written in C (parse.c: 10732 lines, parse.y: 6071 lines). For getting an understanding of the language it is not so handy to read a few thousend lines of C code. We found a statement from the creator of Ruby while searching other resources for a grammar of the Ruby language. After reading it we stopped our search and accept that there are some limitations of our approach when parsing source code written in Ruby.

> *"And documentation, currently there's only one documentation, which is written in C. Oh, one more in my brain."* – Yukihiro Matsumoto, a.k.a Matz (The Creator of Ruby)

Chapter 5

Discussion

5.1 General Statements

When we want to get a quick overview of a system that we do not know at all, our approach works almost perfectly. We can get some information without the need for writing a parser by hand. If we cannot parse a whole system we get feedback about the errors. So, we know which part of the system we cannot parse and can estimate if the model is usable although this part is missing in the model.

There is also a psychological effect in our approach: While working with *CodeSnooper* for a while the user gains a feeling for specifying examples. He learns which examples are meaningful. After a short time period the user tries to get as much from a system as he can. Because he can change the examples himself and compare the results from different parsers almost immediately he is encouraged to get better examples.

With our approach it is possible to build a model from a system in an iterative way. If we want to translate a big system into a model we can start with a small example. When we look at the results we can use that as feedback for going back and specify different or more examples. The whole composition of a fully working parser is a process of going back and forth all the time. This has the positive side effect that we learn something about the system we want to translate already while translating it. We get a feeling for the system already when we parse it because we have to work with the code. As an example we can look at our Java case study (see also Section 4.1). In a first pass we just specify the mapping from a Java interface to a *Class* and look if the parser works. Even if we do not care about the direct result of that run we know the approximate number of interfaces in this project. When we have a parser that fulfills our requirements in the end and export a model, then we already have some basic knowledge about that model. We can then start to work with it more efficiently.

5.2 Problems and Possible Solutions

In the following sections we discuss some problems of our approach and possible solutions to them:

1. Section 5.2.1: Ambiguous grammars
2. Section 5.2.2: False positives and false negatives
3. Section 5.2.3: One grammar is not enough
4. Section 5.2.4: Language constructs in unexpected contexts

5. Section 5.2.5: Complex control structures
6. Section 5.2.6: Packages in Java
7. Section 5.2.7: Inner classes

5.2.1 Ambiguous Grammars

Problem The problem we describe here was the hardest one for us to solve. There may be very simple examples of source code to model mappings that lead into an ambiguous grammar. Because we use SmaCC for the generation of our parsers it is possible to use grammars that are ambiguous - but the result that we get from such a parser is not the one that we expect in most of the cases. So, for this work, we only use grammars that are not ambiguous. If the grammar that is generated based on our examples is ambiguous *CodeSnooper* just refuses to continue.

Here are two cases to illustrate when and why a grammar becomes ambiguous.

- Excluding a keyword in a context where we have a production that uses this keyword for its definition leads in most cases to an ambiguous grammar. Generally speaking it is difficult to see which keywords we have to ignore in which contexts.
- One source for ambiguous grammars is described in Section 3.6.4. When we merge productions that are generated based on different examples it is often not obvious for the user why the grammar becomes ambiguous.

Solution It is difficult to debug an ambiguous grammar. Sometimes it is obvious why the grammar is ambiguous and we can just exchange an example with another one. But in other cases it is difficult to find the source for the ambiguities.

We have two possibilities to go on when we have an ambiguous grammar:

1. We can specify other examples and try different combinations of examples.
2. We can use the same solution that we use for solving the problem where one grammar is not enough to parse a whole system. We refer to Section 5.2.3 for a detailed description of that problem. While working with *CodeSnooper* we learn that it is often useful to use more than one grammar. Normally we get very quickly a grammar that works for a big part of a system. It is difficult to find the right examples to complete this grammar to make it work for the whole system. We see that the grammar becomes ambiguous and difficult to debug. We can just take some files that we cannot yet parse and specify examples in those files until we get a grammar that works with some of these files. If we get a grammar that is ambiguous and we do not know why we ignore it and go to another file. The chances are good that we find a parser that can also parse the files for which we got an ambiguous grammar.

Our experience shows that we can go on quickly when working with the second possibility. We get a grammar that works with around 50-70% of a system with only a few examples. Then we go into some of the other files and produce some more parsers. We come up to about 80-90% of parsed files with a passable effort. The remaining 10-20% are special things in most cases. If we need them in our model we can specify the elements we want as examples and take them into our model without parsing these files.

5.2.2 False Positives and False Negatives

Definitions **A false positive** is a part of our source code that is recognized by our parser, but which does not correspond to a construct of interest.

A false negative is a part of our source code that is a proper construct of interest, but which is not recognized by our parser.

Problem Because our generated parsers are not precise parsers it can happen that they detect the wrong things while processing the input. This problem often occurs if we have a keyword that may be used for specifying different entities and we do not want to detect all of these entities. For example in Java we have the keyword "**public**". We use this keyword in the definitions of *Classes*, *Methods* and *Attributes*.

Example We look at the following two Java *Classes*:

```
class AJavaClass {
    private static String hello = "Hello World!";
    public static void hello() {
        System.out.println(AJavaClass.hello);
    }
}

public class AnotherJavaClass {
    public static final AnotherJavaClass anAttribute =
        new AnotherJavaClass() {
            public String getHello() {
                return "Hello Inner World!";
            }
        };
    public static final String hello = "Hello World!";
    static void hello() {
        System.out.println(AnotherJavaClass.hello);
    }
    public String getHello() {
        return AnotherJavaClass.hello;
    }
}
```

We specify examples in the first *Class* and want to detect *Classes* and *Methods* but leave *Attributes* out. Depending on the way we specify them we can get the following grammar:

```
Class  ::= "class" <IDENTIFIER> "{" Method* "}"          ;
Method ::= "public" not_known* <IDENTIFIER> "(" ")"
           "{" not_known* "}"                             ;
```

With a parser based on this grammar we detect 3 *Methods* in the second *Class*:

```
public static final AnotherJavaClass anAttribute =
    new AnotherJavaClass() {
        public String getHello() {
            return "Hello Inner World!";
        }
    };
-----------------------------------------------------
public static final String hello = "Hello World!";
static void hello() {
    System.out.println(AnotherJavaClass.hello);
}
-----------------------------------------------------
public String getHello() {
    return AnotherJavaClass.hello;
}
```

The first one is a *false positive*: It is an *Attribute* that is recognized as a *Method*. The second detected *Method* consists of an *Attribute* and a real *Method*. The third one is the only *Method* that is recognized in the correct way.

Solution There is no possibility to find out automatically if our parser detected the wrong things. If we could do that we could also fix the parser so that it does not detect the wrong things anymore. So, we absolutely need user interaction if this problem occurs. The only solution is that we specify more or other examples. But even then it is not guaranteed that we do not run into this problem again.

5.2.3 One Grammar is not Enough

Problem If we provide more than one mapping example for the same target element with conflicting definitions we cannot work with one single parser anymore. This problem arises because we have a language independent model.

Example We can illustrate this problem with a *Java Class* and a *Java Interface*. We map both of them to a *FAMIXClass*:

```
class JavaClass {
    public void hello() {
        System.out.println("Hello World!");
        System.exit(0);
    }
}

interface JavaInterface {
    public void interfaceMethod();
}
```

If we define the mapping examples for that source code we get the following two grammars:

```
Class  ::= "class" <IDENTIFIER> "{" Method* "}"          ;
Method ::= "public" "void" <IDENTIFIER> "(" ")"
           "{" not_known* "}"                            ;

Class  ::= "interface" <IDENTIFIER> "{" Method* "}"      ;
Method ::= "public" "void" <IDENTIFIER> "(" ")" ";"      ;
```

If we merge these definitions we receive the following grammar:

```
Class  ::=   "class" <IDENTIFIER> "{" Method* "}"
           | "interface" <IDENTIFIER> "{" Method* "}"    ;
Method ::=   "public" "void" <IDENTIFIER> "(" ")"
             "{" not_known* "}"
           | "public" "void" <IDENTIFIER> "(" ")" ";"    ;
```

With this grammar we could generate a parser which would build a valid model from the following source code that is not correct Java code:

```
interface JavaInterface {
    public void hello() {
        System.out.println("Hello World!");
        System.exit(0);
    }
}
```

Solution This problem arises only in the cases where we map more than one different language element on one single element in the FAMIX model. In *CodeSnooper* we provide a solution where we use more than one parser for such cases. We look at the examples that the user gives in different files of source code independently and generate a grammar for each example. Then we use each grammar as basis for one parser. So, we get a list of different parsers. We try to parse every file with every parser until one of them is successful. If we look at the example above we can see that with different parsers (one based on the first grammar and another one based on the second grammar) we get the possibility to map different language elements on one singe model element. If we do not mix different elements in one single example then our approach with more than one parser never produces a wrong model.

Another possible solution is to take an intermediate language specific model as our target. Then we could make a model transformation from the language specific model to the language independent FAMIX model.

5.2.4 Language Constructs in Unexpected Contexts

Problem For describing this problem we just speak of tokens. With this term we identify tokens from the scanner (see Section 3.1) and also *LanguageConstructs* as described in Section 3.5.

Whenever our parser finds a token in the source code it must decide what to do with it. This decision can be wrong when a token appears in a context where we do not want it to be recognized as a token.

Example We look at the following lines of Java code:

```
System.getProperty(SOAPFactory.class.getName(),DEFAULT_FACTORY);

static final String UNIBE_URI="http://www.unibe.ch/";
```

When reading the first line our parser fails because the token "**class**" is in a context where our parser does not expect it to be. We use this token for the definition of a *Class*, but here there is no such definition.

On the second line everything works fine until our parser reads the double slash "**//**". Because of our scanner definition this is the start token of a single line comment. The parser skips now everything until the end of the line. So, we also miss the semicolon and the definition of this attribute is not complete.

Solution There is more than one possibility to solve that problem:

- We can decide to ignore the problematic tokens in the particular context. This works for solving the problem with "**class**" token. The drawback is that we probably lose information. In this example it is not possible to detect *Inner Classes* anymore after adding the "**class**" token to the *ignore* tokens in the context of a *Class*. This solution alone does not work for the second example.
- We can refine our tokens. For solving the problem with the start token of a comment in a string, we can refine the scanner definition in a way that the comment definition does not match if it appears inside a string. Unfortunately we reach the limit of a scanner generated by SmaCC: When the token definitions reach a certain complexity the generated scanner works only with small files. With bigger input files it stops to deliver tokens.
- For solving the problem with language constructs in strings, we can also start to detect strings and work with them as tokens. The drawback is that our grammars become more complex because we have to take care of all strings.

Generally speaking, the two solutions for this problem are: Either we ignore more things or we detect more things. The first solution yields to simple and understandable grammars with the drawback of information loss. The second solution produces more complex grammars that are more difficult to maintain. In exchange for that we get more precise information about the system.

5.2.5 Complex Control Structures

Problem With our implementation of *LanguageConstructs* we can handle keywords and simple block constructs (see also Section 3.5). This is not enough for parsing languages that support more complex control structures.

Example Here are two code snippets written in Ruby:

```
case str
when "0" then false
when "1" then true
else
  raise "something is wrong"
end
-------------------------------------------------------------
return true if(@filters.empty?)
@filters.each do |filter|
  result = filter[test]
  if(result.nil?)
    next
  elsif(!result)
    return false
  else
    return true
  end
end
```

First of all, we must care about such language constructs in this case because the keyword "**end**" is also used in the definition of *Modules*, *Classes* and *Methods*. If we analyze these examples we find the following control structures:

```
case ... when ... then ... else ... end
return ... if(...)
do ... end
if(...) ... elsif(...) ... else ... end
```

Besides the fact that these structures are quite complex the brackets are sometimes optional and sometimes they are required. So, we are just not able to work with these structures with our implementation of *LanguageConstructs*.

Solution There is only one solution to this problem: we must extend the definition of our *LanguageConstructs* to handle such complex structures. This leads to more complex grammars and maybe also to more ambiguities (see also Section 5.2.1). If we split these constructs up in simpler ones and combine them in some clever grammar rules this problem is solvable in an elegant way.

5.2.6 Packages in Java

Problem There are a lot of *Classes* that have the same name but are not in the same package of course. If we do not detect packages, then we have overlapping entities in our model. So, it is possible that *Methods* and *Attributes* are added to the wrong *Class*-entity in our model.

Solution There are some possibilities to solve that problem:

- We can isolate each package and look at it on its own.
- We can start to detect packages as well but then, there are other problems that arise. We should be able to generate FAMIXEntities on the fly. This would be future work and is mentioned in Section 6.3.
- We can do a trick with namespaces. In Java the package definitions correspond to the folders on the filesystem. We can just add a placeholder for the namespace generated from the path of the folder in which the file is and add any detected entity from this file to that placeholer. That is the solution we use for the Java case study.

5.2.7 Inner Classes

Problem We cannot specify an example of an *Inner Class* because that leads to a wrong model. When exporting a model from a tree we assume that a *Node* always belongs to its parent *Node*. That is not quite true for an *Inner Class*. An *Inner Class* does not belong to its surrounding *Class*. It has its own name and belongs to the package to which also the surrounding *Class* belongs.

Solution We do not have a solution for that problem in our implementation. However it can be solved by allowing to set relationships between the *Nodes* in our tree not according to the hierarchy: In our tree the *Inner Class* would be a child of the surrounding *Class* but we set a relationship between the *Inner Class* and the package to which also the surrounding *Class* belongs. We include this idea in Section 6.3.

5.3 Benefits of our Approach

We summarize some benefits of our approach:

- **Hiding of complexity**: Normally we know some things about the systems we want to transform. But probably we are not used to build parsers. With our approach we get a possibility to build parsers based on the knowledge that we have. If the result does not seem right to us, we can refine the examples and try again - without the need to work with grammars and parsers directly: It is a much more intuitive way to work. Nevertheless we have the possibility to adjust every step while generating parsers.
- **Interactivity**: We always see the consequences of our actions. When we specify an example and the generated parser does not work anymore we know that this example is the cause. The generation of a useful parser is a process of going back and forth: We get a feeling for specifying proper examples.
- **Flexibility**: With our approach we can handle any input. As soon as we recognize our target elements in the input we can generate a model from that input. We can even exchange the target model if needed.
- **Working with a system**: While specifying examples and trying different combinations of examples we get some information about the input. When we have a model of that system we do already have some basic knowledge about that model and can start to work with it more efficiently.

5.4 Limitations of our Approach

Our approach does also have some limitations. We could probably eliminate some of them with another implementation.

- **Precise parsing**: We cannot do precise parsing but that was not an intended goal of our approach. However, we can never eliminate any possibility of false positives and/or false negatives. If we absolutely need a perfect model of a system we have to write a precise parser.
- **Well chosen examples**: If we want to get a lot of information from a system we often run into problems with ambiguous grammars. It is possible to get almost all information from a system with our approach but for that we need a lot of well chosen examples. Depending on the input it is difficult to pick the right examples. If we cannot recognize our target elements in the input we do not get any information from that system.
- **Ambiguous grammars**: Often, we get good results quite quickly. But we can also run into problems just as quickly. We specify some examples and build a grammar which turns out to be ambiguous. If we do not recognize immediately what we have done wrong it may be difficult to debug this grammar/examples.
- **Relationships between nodes**: With our approach it is difficult to model information like *Invocation* and *Access*. It is much more easy to work with *Class*, *Method* and *Attribute* definitions, because here, we have a hierarchy between the elements. To model *Invocation* and *Access* we need more complex relationships between the *Nodes* than just the parent-child-connection. However, it should be possible to extract such information with our approach. But our *Nodes* must support optional relationships and the user interface must be improved. See also Section 6.3 for more ideas.

Chapter 6

Conclusion and Future Work

6.1 Parsing by Example compared with other Approaches

With our approach the user has control over all the steps that are needed to build a parser. If he wants he can even modify the grammar before the generation of the parser. But if the user does not have the knowledge about grammars and parsers he does not need to do so. The only thing that the user needs to do is to write a regular expression for recognizing identifiers and comments in the target programming language as written in Section 3.1. There are a lot of those definitions available - this is not a drawback of our approach.

Parsing based on examples is a mixture of other approaches. It reduces the drawbacks of other approaches and increases their advantages. The biggest advantage of our approach is that we make parsing transparent to the end user. We can modify almost everything and tune every single step. With our approach one can produce different parsers in a short time - just specify examples, generate a parser, look at the result, redefine examples, generate a new parser and look at the differences. This is almost impossible with other approaches because grammar rules are too difficult to maintain in short time intervals.

In Section 2.3 we mentioned some problems that other approaches have. Now, we look at them again and discuss if we can solve them with our approach.

- **Generalization is missing:** Our approach provides a solution to that problem. Most approaches only work with a particular type of input and provide only a special model as output. Our approach is absolutely independent from the input and is flexible enough to work with different target models. We can switch the target model without a big effort. This is a drawback of some other approaches that are fixed to a particular target model.

- **No error recovery:** We do not solve this problem completely, but we help the user to recover from an error. If we have a running parser but cannot parse the whole input we just skip a file and go on with the next. We give immediate feedback to the user while parsing. He has always a list of successfully parsed files. Because we work in an iterative way the user normally detects faults early.

  With some approaches we get either all or nothing. If parsing fails somewhere the whole output is discarded. This is different with our approach. We can also generate a model from partially parsed code.

- **Context is not taken into account:** With our approach this problem can be solved completely. But in the reference implementation we only solved it partially. We only use the information we have in our trees for generating grammars and parsers. Right now, our *Nodes* know their children and their parents but we cannot define arbitrary relationships between *Nodes* in the tree. We added this also to the section about future work (see Section 6.3).

6.2 Conclusion

Normally, the people that work in the field of software reengineering do not have a deep knowledge about grammars and parsers. They are experts in analyzing models. They need support from a tool to extract a model from the system they want to analyze. Our approach proved that it is possible to extract partial models from arbitrary source code without deeper knowledge about parsing techniques.

Even if we have an approach that really works there is one last challenge: to provide a easy to use graphical interface. With *CodeSnooper* we have also a proof of concept that it is possible to create an user-friendly interface. *CodeSnooper* is a proof of concept that we could improve in many ways. We refer to Section 6.3 for some ideas.

If we look at all the reengineering tools that are available and all languages they support we state that there is a lack of easy to use parser generators. This circumstance alone shows the importance of this work. We are sure that the ideas discussed in this work can help to develop easy to use source model extractors for reverse engineering.

6.3 Future Work

There are many things that we could do based on our approach. We just name a few of them here:

- **Extend the target model**: We could extend our target model on the fly. If we provide an example of a new element we could generate a new target for that element and in this way extend the model.
- **Artificial Intelligence**: There are two parts in our approach where we could use artificial intelligence:
    1. We could use pattern recognition in combination with our method of grammar generation. So, we could probably detect some ambiguities in the grammar without interaction of the user.
    2. We could also try to use methods from the field of artificial intelligence directly on the source code (pattern recognition, inference, learning, planning, heuristics, ...) in order to need even less examples to generate a useful grammar. If this works we could even try to parse 'the bible' and transform that into a model.

  We do not encourage anyone to work with artificial intelligence only and without examples at all. If we would only use artificial intelligence to detect patterns in the code and try to map them to our model entities, this would have the following drawbacks:

    – It would be very hard to fix the application if some results are not correct.
    – For the user it would be unreproducible what the application really does and it would be hard to understand why some entities are recognized while others are not.
- **Central grammar database**: We could store all generated grammars in one centralized database. So, we could access also the grammars that someone else has used in some other place.
- **Store examples**: It would be useful not only to store the parser definitions as explained in Section 3.10 but also the examples that the user makes.

- **Drag and Drop**: For the recognition of things like inheritance and invocation it would be nice to have some drag'n'drop functionality. So, the user could express such connections in a easy way. But this would lead to a more complex user interface (see also the next point). We could also use drag'n'drop functionality to express the belongsTo-relationship between *Nodes* not according to the hierarchy in our tree. In order that this would work we must also extend our *Nodes* in a way that they support such relationships.

- **Visualization of nodes**: Our example and parse trees could be visualized as a graph. We could even use CodeCrawler as visualization tool [LANZ 05]. With such a visualization it should be possible to manage drag'n'drop functionality. It would be also be possible to color *Nodes* based on the grammar productions that were involved in the *Node* generation. This would help a lot if we have some false positives or false negatives in our parse tree.

- **Specify examples on parsed code**: Because our example and parse trees are composed of the same *Node* elements it would be possible to specify examples on already parsed code. There are some problems that arise because we have too much information in such a parse tree. It would be difficult (but needed) to filter the generated rules.

Appendix A

Tools

A.1 CodeSnooper

In this appendix we give a brief overview of the tool *CodeSnooper*, the reference implementation of our approach. We describe the steps that are needed to get a running version of *CodeSnooper*. We look at the requirements and show the different parts of the application. Chapter 3 contains a detailed description of all steps from source code to a valid model. In this appendix we look at *CodeSnooper* from the perspective of a user.

A.1.1 Quick Start

We use Cincom VisualWorks NonCommercial, 7.2[1] for the development of *CodeSnooper*. We provide the instructions here based on this version.

1. Start VisualWorks with a clean image.
2. Connect to the Store of the Software Composition Group.
3. Load the bundle called *CodeSnooper*. Some prerequisites are loaded automatically before *CodeSnooper* itself:
   - Moose: The latest version we tested with *CodeSnooper* for this work is (3.0.18,girba). *CodeSnooper* should also run with newer versions of MOOSE. There is only a small interface between *CodeSnooper* and MOOSE. The whole connection to MOOSE is done by the following three classes: MooseImporter, Target and TargetAttribute.
   - SmaCC: We work with version (1.3Cincom,wuyts).
   - Method Wrapper Base: Here we use version (1.1,greevy).
4. Start *CodeSnooper*: There is a new entry in the Tools menu.
5. Choose ‘Load SourceCode from Files...’ from the File menu of *CodeSnooper* and select one or more source files to work with.
6. Start with the definition of a *Class* by highlighting the corresponding part of the source code. Open the pop-up menu by clicking the right mouse button and choose *Class* from it. *CodeSnooper* generates a new *Node* in the list on the right side of the main view. The color of the *Node* changes from red to green as soon as there is a name specified for it. Specify the exact signature of this *Class* (only keywords and the name of the *Class* are allowed in this signature).

[1] `http://smalltalk.cincom.com/`

7. Choose 'Manage Keywords' to define which *LanguageConstruct* should be ignored in which contexts.
8. Click 'Build Grammar' to generate a grammar based on the example of the *Class*.
9. Build a parser and try to parse the file with it by clicking the 'Parse File' button. *CodeSnooper* gives feedback if the actual source code could be parsed or not.
10. Choose 'Manage Parser Definitions...' from the File menu to load or to save a parser definition. We need a connection to the internet to load a parser definition from an XML file. Otherwise we cannot find the DTD that we need to validate the XML file.

A.1.2 Requirements

Besides a working implementation of our approach, we have the following requirements for our application:

- **User-friendly interface**: It must be possible to specify mapping examples in an intuitive way. A user without knowledge about parsing should be able to work with *CodeSnooper* after a short learning phase. Nevertheless it should be possible to tune every step for another user that has knowledge about parsing.
- **Reuse parser definitions**: When we have a working parser we want to store its definition for later use.
- **Error handling**: The user must get useful error messages if something goes wrong. *CodeSnooper* colorizes the source code while parsing when it detects a token that it does not expect in that place. The user can choose if he wants to get error feedback or not.

A.1.3 User Interface

Main View

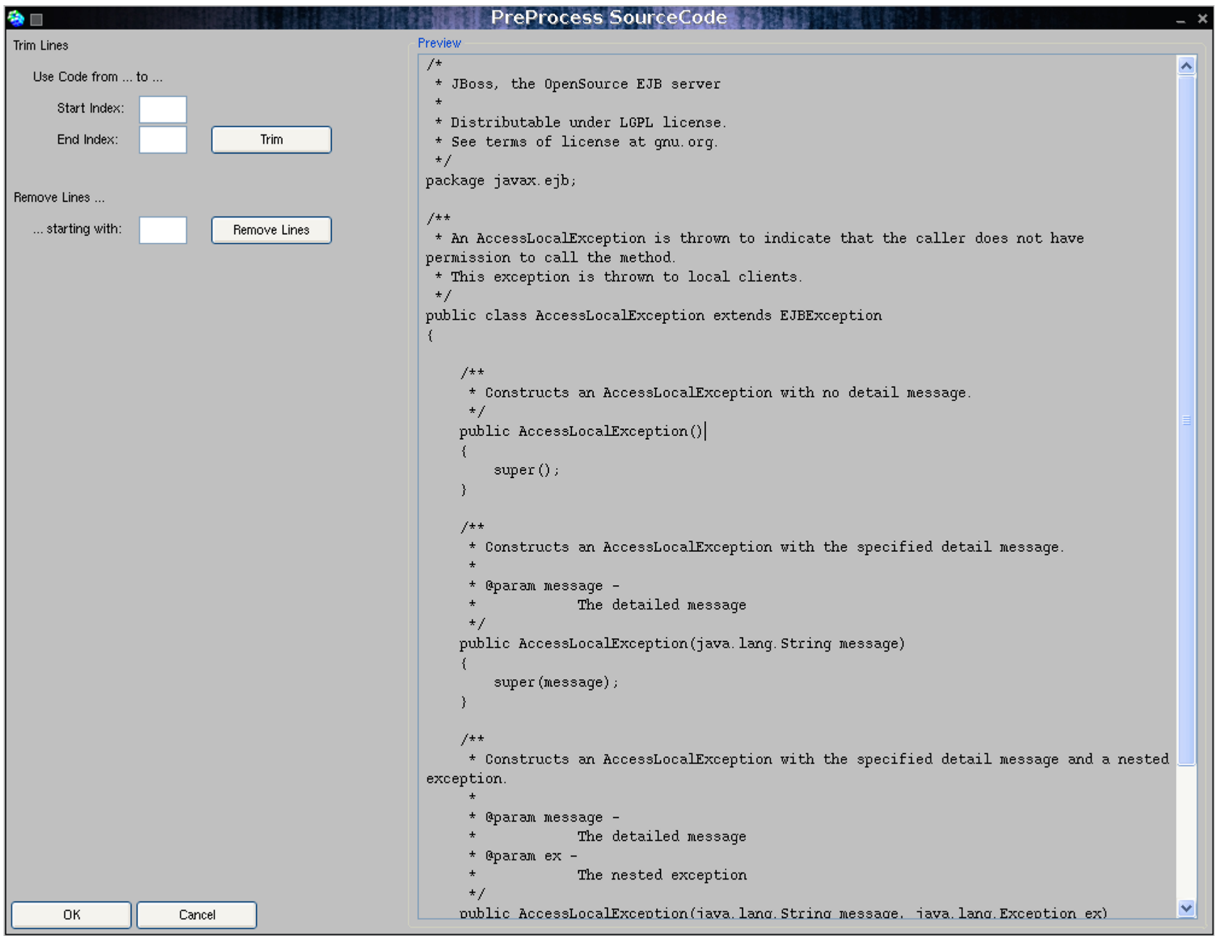
In this work we have two screenshots of the main view of *CodeSnooper*:

1. Figure 4.1 shows the main view while specifying an example. We have the source code of the current *Node* in the middle of the view. The pop-up menu that we get when clicking the right mouse button consists of two parts:
   - A context insensitive part: With this instructions we can create new *Nodes*, give a name to the actual *Node* or specify new *LanguageConstructs*.
   - A context sensitive part: With the instructions in this part, we set the attributes for a *Node*. It depends on the target of the *Node* which attributes are available.
2. Figure 4.2 shows the main view after parsing the whole input. On the left side we see which files are successfully parsed (the green ones). On the right side we see which elements are detected in the actual selected source file.

Preprocessor

Figure A.1 shows the graphical user interface of our preprocessor. We used the preprocessor for one single example during the development of *CodeSnooper*. It was an example of COBOL code where we had to do some preprocessing of the code: We had to truncate every line by a few characters at the beginning and at the end. So, we just implemented two features:

- We can cut out a specified number of characters from every line of the source code that we have as input.
- We can delete every line that starts with a certain string.



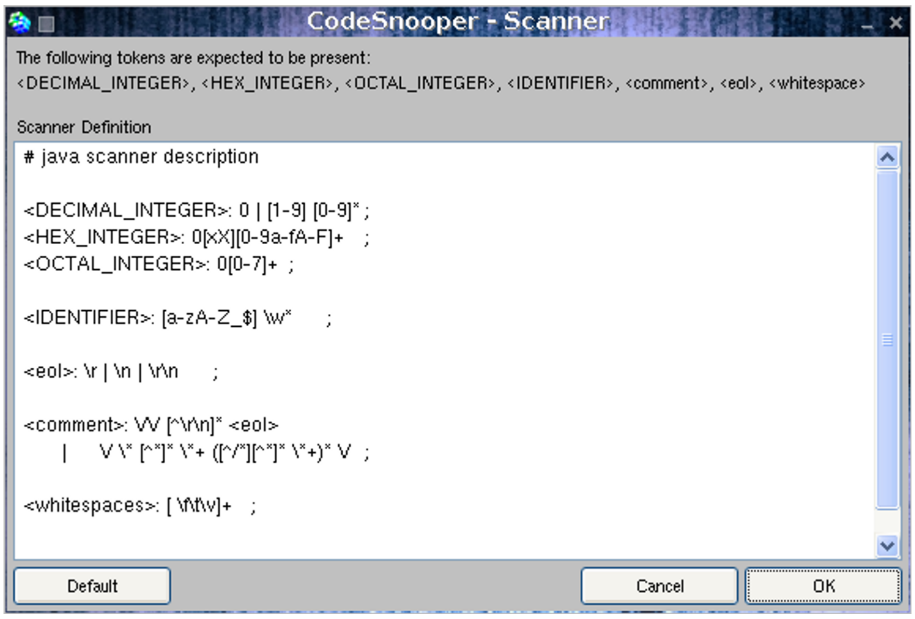
```
/*
 * JBoss, the OpenSource EJB server
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */
package javax.ejb;

/**
 * An AccessLocalException is thrown to indicate that the caller does not have
permission to call the method.
 * This exception is thrown to local clients.
 */
public class AccessLocalException extends EJBException
{

    /**
     * Constructs an AccessLocalException with no detail message.
     */
    public AccessLocalException()
    {
        super();
    }

    /**
     * Constructs an AccessLocalException with the specified detail message.
     *
     * @param message -
     *            The detailed message
     */
    public AccessLocalException(java.lang.String message)
    {
        super(message);
    }

    /**
     * Constructs an AccessLocalException with the specified detail message and a nested
exception.
     *
     * @param message -
     *            The detailed message
     * @param ex -
     *            The nested exception
     */
    public AccessLocalException(java.lang.String message, java.lang.Exception ex)
```

Figure A.1: CodeSnooper: User interface of the preprocessor.

Scanner and Grammar Views

Figure A.2 shows the interface for defining scanner tokens. We do not process this definition in any way: We give it to SmaCC as it is written here. SmaCC performes a syntax check of this definition when we generate a parser.

We show the interface for analyzing and modifying the generated grammar in Figure A.3. We can modify, add and remove productions from the grammar. We can even change the reduce actions for the productions although this is normally not needed.

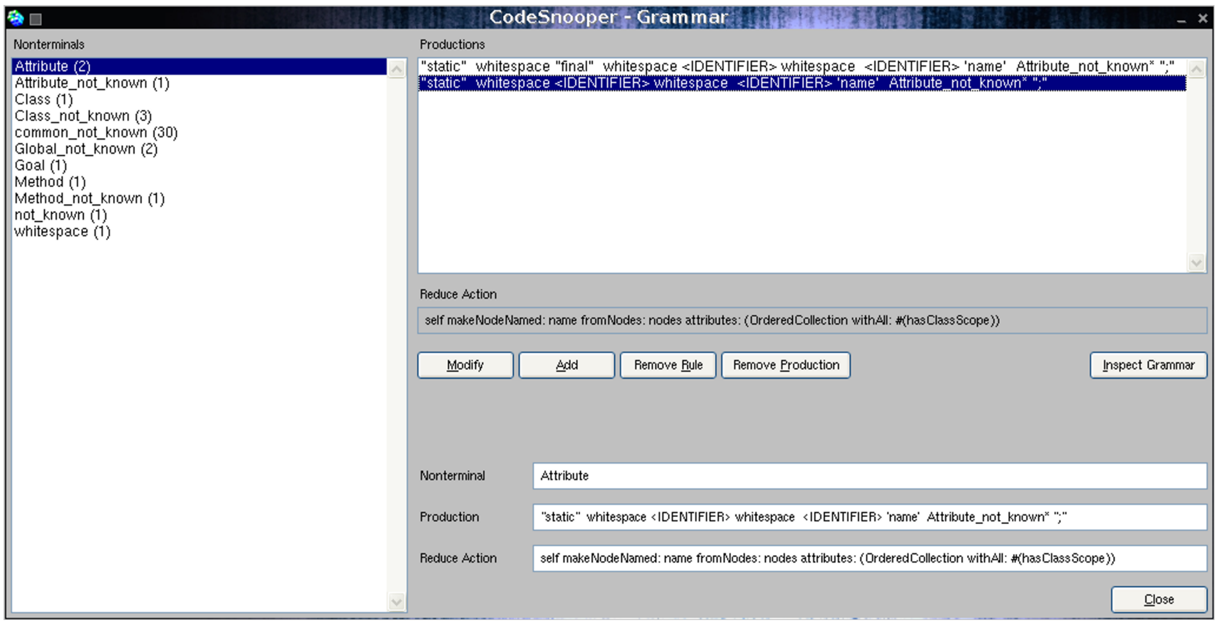


The following tokens are expected to be present:
<DECIMAL_INTEGER>, <HEX_INTEGER>, <OCTAL_INTEGER>, <IDENTIFIER>, <comment>, <eol>, <whitespace>

Scanner Definition

```
# java scanner description

<DECIMAL_INTEGER>: 0 | [1-9] [0-9]* ;
<HEX_INTEGER>: 0[xX][0-9a-fA-F]+    ;
<OCTAL_INTEGER>: 0[0-7]+  ;

<IDENTIFIER>: [a-zA-Z_$] \w*      ;

<eol>: \r | \n | \r\n      ;

<comment>: \/\/ [^\r\n]* <eol>
     |     \/ \* [^*]* \*+ ([^/*][^*]* \*+)* \/  ;

<whitespaces>: [ \f\t\v]+    ;
```

Figure A.2: CodeSnooper: User interface for changing the scanner definitions.



Nonterminals

- Attribute (2)
- Attribute_not_known (1)
- Class (1)
- Class_not_known (3)
- common_not_known (30)
- Global_not_known (2)
- Goal (1)
- Method (1)
- Method_not_known (1)
- not_known (1)
- whitespace (1)

Productions

```
"static"  whitespace "final"  whitespace <IDENTIFIER> whitespace  <IDENTIFIER> 'name'  Attribute_not_known* ";"
"static"  whitespace <IDENTIFIER> whitespace  <IDENTIFIER> 'name'  Attribute_not_known* ";"
```

Reduce Action

self makeNodeNamed: name fromNodes: nodes attributes: (OrderedCollection withAll: #(hasClassScope))

Nonterminal: Attribute

Production: "static" whitespace <IDENTIFIER> whitespace <IDENTIFIER> 'name' Attribute_not_known* ";"

Reduce Action: self makeNodeNamed: name fromNodes: nodes attributes: (OrderedCollection withAll: #(hasClassScope))

Figure A.3: CodeSnooper: User interface for modifying the grammar.

Language Constructs Manager

With the support of the manager showed in Figure A.4 we define which *LanguageConstructs* we want to ignore in which contexts. On the left side we have all available contexts. Two of them are special ones:

- **Common**: The *LanguageConstructs* that we ignore in this context are the things that we want to ignore in all contexts.
- **Global**: This is the context that surrounds the top *Node*. For example in a Java source file this includes the package definition when we specify a *Class* as top *Node*.

We generate the context specific ignore rules based on the information from this interface.

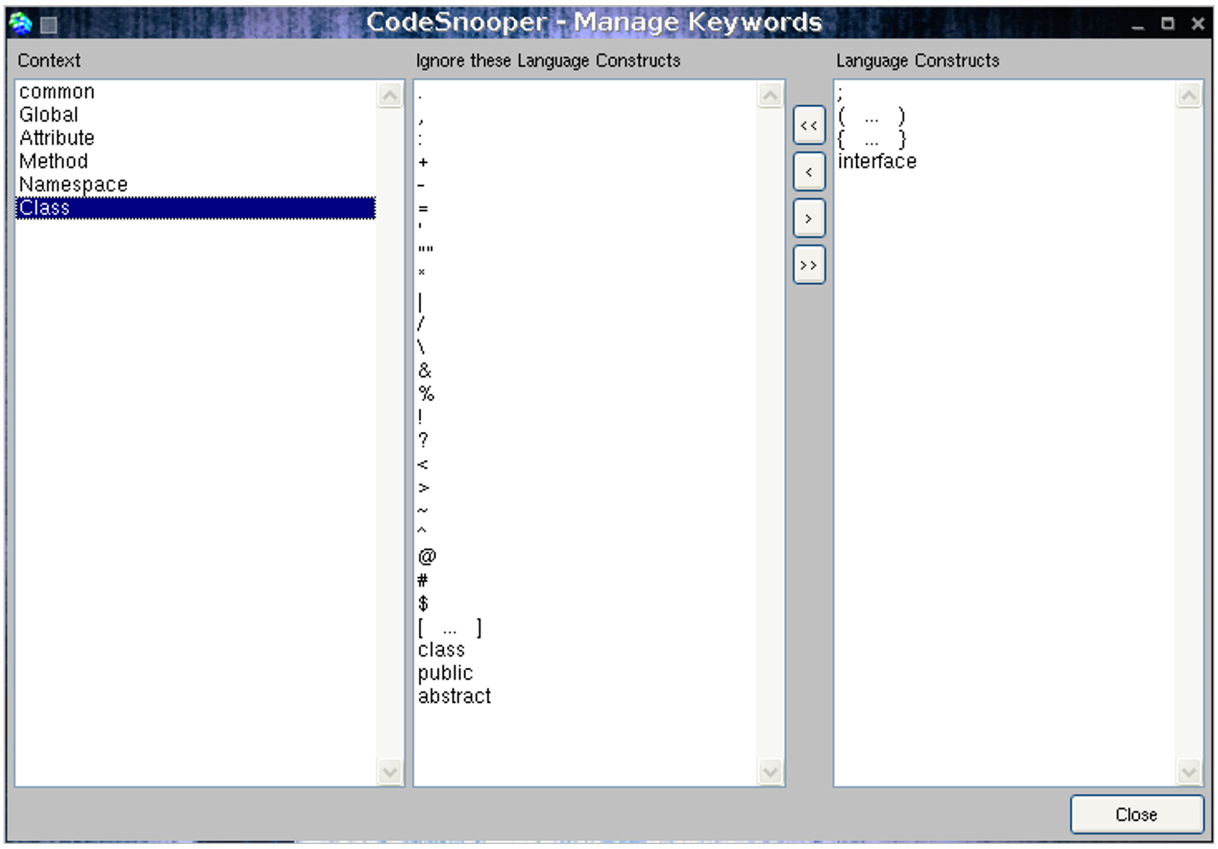


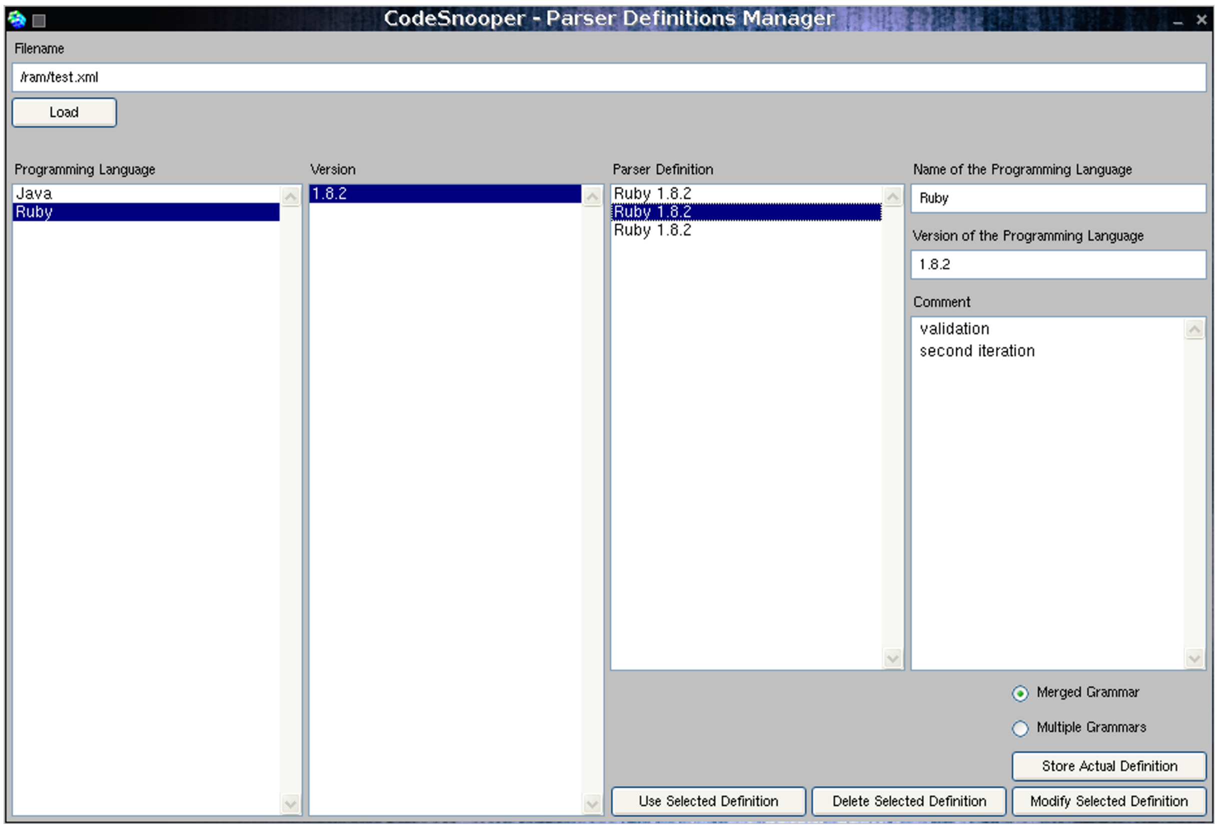
Figure A.4: CodeSnooper: User interface for managing language constructs.

Parser Definitions Manager

Figure A.5 shows the parser definitions manager that we use to store and load parser definitions. We can store any numer of definitions in the same file according to the document type definition shown in Appendix B.1.

With the parser definitions manager we can delete existing definitions or modify their data. We can also add the parser definition that we use at the moment to a file.

When we select an existing definition and want to parse our input based on this definition we just click on 'Use Selected Definition': A grammar is rebuilt from that definition and together with the scanner from that definition we rebuild a parser and use it to parse all of the input files.



Filename: /ram/test.xml

Programming Language: Java, Ruby (selected)

Version: 1.8.2 (selected)

Parser Definition: Ruby 1.8.2, Ruby 1.8.2 (selected), Ruby 1.8.2

Name of the Programming Language: Ruby

Version of the Programming Language: 1.8.2

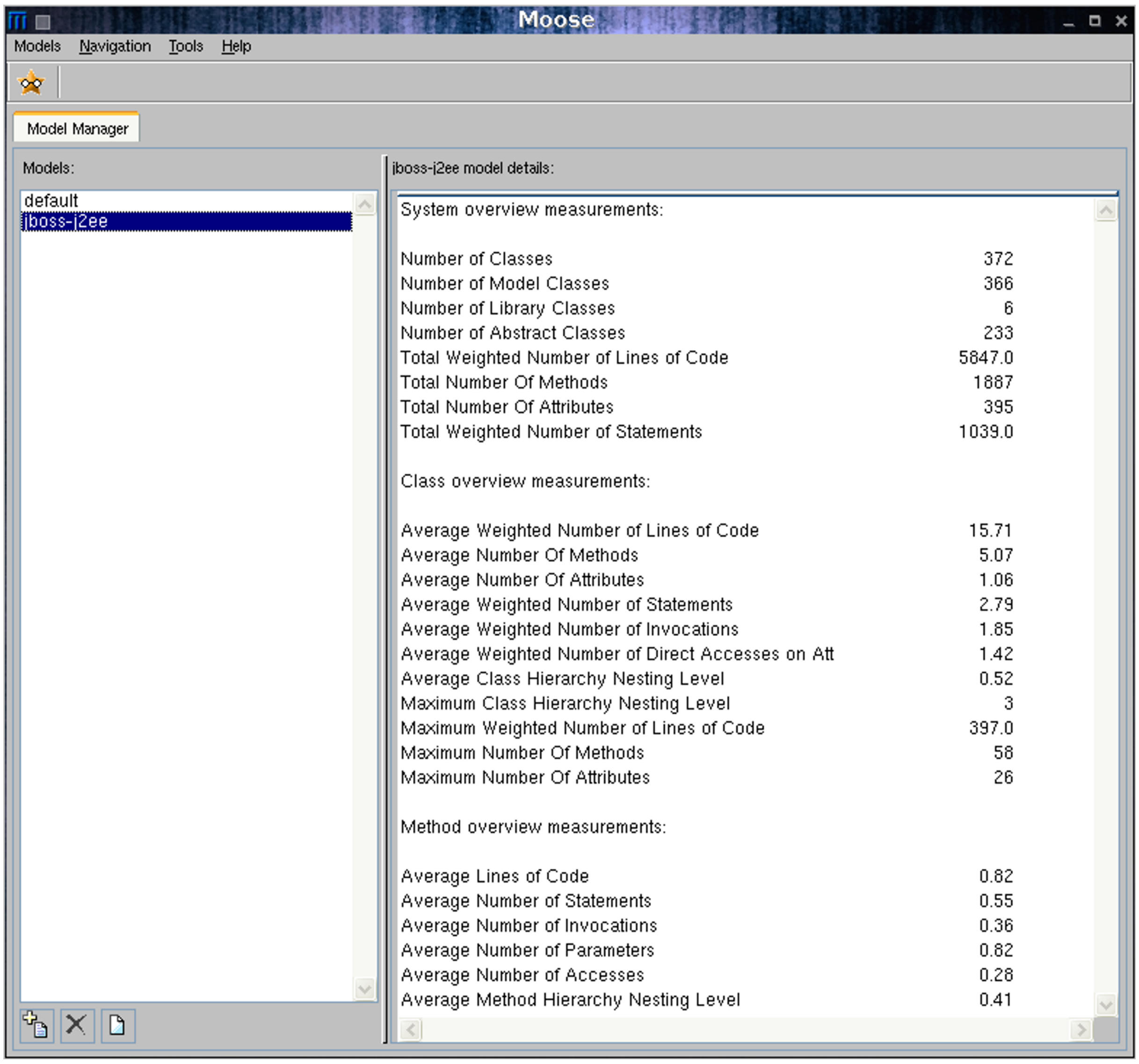
Comment:
validation
second iteration

[✓ Merged Grammar] Multiple Grammars

Figure A.5: CodeSnooper: User interface for managing parser definitions.

A.2 MOOSE

We used the MOOSE reengineering environment for validating our approach. In Figure A.6 we see the main view of MOOSE. We loaded a model of the jboss-j2ee package that we used for our Java case study into MOOSE.



```
System overview measurements:

Number of Classes                                        372
Number of Model Classes                                  366
Number of Library Classes                                  6
Number of Abstract Classes                               233
Total Weighted Number of Lines of Code                5847.0
Total Number Of Methods                                 1887
Total Number Of Attributes                               395
Total Weighted Number of Statements                   1039.0

Class overview measurements:

Average Weighted Number of Lines of Code               15.71
Average Number Of Methods                               5.07
Average Number Of Attributes                            1.06
Average Weighted Number of Statements                   2.79
Average Weighted Number of Invocations                  1.85
Average Weighted Number of Direct Accesses on Att       1.42
Average Class Hierarchy Nesting Level                   0.52
Maximum Class Hierarchy Nesting Level                      3
Maximum Weighted Number of Lines of Code               397.0
Maximum Number Of Methods                                 58
Maximum Number Of Attributes                              26

Method overview measurements:

Average Lines of Code                                   0.82
Average Number of Statements                            0.55
Average Number of Invocations                           0.36
Average Number of Parameters                            0.82
Average Number of Accesses                              0.28
Average Method Hierarchy Nesting Level                  0.41
```

Figure A.6: MOOSE: Main view with one loaded model.

Appendix B

Definitions

B.1 Document Type Definition for our Parser Definitions

The XML files in which we store our parser definitions must be valid instances of the following document type definition (DTD):

```
<!ELEMENT codesnooper (parser+)>
<!ELEMENT parser (scanner,grammar+,comment?)>
<!ELEMENT scanner EMPTY>
<!ELEMENT grammar (rule+)>
<!ELEMENT rule (production+)>
<!ELEMENT production EMPTY>
<!ELEMENT comment EMPTY>

<!ATTLIST parser language CDATA #REQUIRED>
<!ATTLIST parser version CDATA #IMPLIED>
<!ATTLIST scanner definition CDATA #REQUIRED>
<!ATTLIST rule nonterminal CDATA #REQUIRED>
<!ATTLIST production rule CDATA #REQUIRED>
<!ATTLIST production reduceAction CDATA #IMPLIED>
<!ATTLIST comment content CDATA #REQUIRED>
```

B.2 Validation Environment

Here we show an overview of the setup of the computer we worked with during the case studies.
Hardware:

- AMD Athlon XP 2600+
- 1GB RAM

Operating System:

- GNU[1]/Linux (Gentoo[2])
- Kernel 2.6.10-mm1[3]

Software:

- Cincom VisualWorks NonCommercial, 7.2[4]
- Sun JDK 1.5.0[5]
- eclipse 3.1[6]
- jFamix 0.7.0[7]
- Ruby 1.8.2[8]
- FreeRIDE 0.9.2[9]

[1] `http://www.gnu.org/`
[2] `http://www.gentoo.org/`
[3] `http://kernel.org/`
[4] `http://smalltalk.cincom.com/`
[5] `http://java.sun.com/j2se/1.5.0/`
[6] `http://www.eclipse.org/`
[7] `http://jfamix.sourceforge.net/`
[8] `http://www.ruby-lang.org/`
[9] `http://freeride.rubyforge.org/`

B.3 Grammar Example

The grammar we show here is in the form like *CodeSnooper* gives it to SmaCC. It is the grammar that we used in the first iteration of the Java case study (detection of *Classes* only).

```
Goal ::=
        (Global_not_known | Class)*
        { self makeNotKnownNodefromNodes: nodes rule: #Goal production: 1 } ;

Class ::=
        "class" whitespace <IDENTIFIER> 'name' not_known* "{" Class_not_known* "}"
        { self makeNodeNamed: name fromNodes: nodes rule: #Class production: 1 }
    |   "interface" whitespace <IDENTIFIER> 'name' not_known* "{" Class_not_known*
        "}" { self makeNodeNamed: name fromNodes: nodes rule: #Class production: 2 }
    |   "abstract" whitespace "class" whitespace <IDENTIFIER> 'name' not_known*
        "{" Class_not_known* "}"  { self makeNodeNamed: name fromNodes: nodes
        attributes: (OrderedCollection withAll: #(isAbstract)) rule: #Class
        production: 3 } ;

Method_not_known ::=
        common_not_known    ;

not_known ::=
        common_not_known    ;

Global_not_known ::=
        common_not_known    ;

Attribute_not_known ::=
        common_not_known    ;

Class_not_known ::=
        common_not_known
    |   "class"
    |   "(" Class_not_known* ")"
    |   "{" Class_not_known* "}"
    |   "[" Class_not_known* "]"
    |   "abstract"  ;

common_not_known ::=
        <DECIMAL_INTEGER>
    |   <HEX_INTEGER>
    |   <OCTAL_INTEGER>
    |   <IDENTIFIER>
    |   <comment>
    |   <eol>
    |   <whitespaces>
    |   "." | "," | ":" | ";" | "+" | "-" | "=" | "'" | "*" | "|" | "/" | "\" | "&"
    |   "%" | "!" | "?" | "<" | ">" | "~" | "^" | "@" | """"    ;

whitespace ::=
        <whitespaces>?  ;
```

List of Figures

List of Tables

76

Bibliography

[BLAS 01] D. Blasband. *Parsing in a hostile world.* In Proceedings WCRE 2001, pages 291–300. IEEE Computer Society, October 2001. (pp 2, 9)

[DEME 01] S. Demeyer, S. Tichelaar, and S. Ducasse. *FAMIX 2.1 — The FAMOOS Information Exchange Model.* Research report, University of Bern, 2001. (pp 11, 23)

[DEME 02] S. Demeyer, S. Ducasse, and O. Nierstrasz. Object-Oriented Reengineering Patterns. Morgan Kaufmann, 2002. (p 1)

[DUCA 05] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. *Moose: a Collaborative and Extensible Reengineering Environment.* In Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series, pages 55 – 71. Franco Angeli, 2005. (pp 11, 23)

[JOHN 75] S. Johnson. *Yacc: Yet Another Compiler Compiler.* Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975. (p 1)

[KLUS 03] S. Klusener and R. Lämmel. *Deriving tolerant grammars from a base-line grammar.* In Proceedings ICSM 2003, pages 179–188. IEEE Computer Society, September 2003. (pp 2, 7)

[LANZ 05] M. Lanza and S. Ducasse. *CodeCrawler - An Extensible and Language Independent 2D and 3D Software Visualization Tool.* In Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series, pages 74 – 94. Franco Angeli, 2005. (p 59)

[LATE 03] M. Latendresse. *RegReg: a Lightweight Generator of Robust Parsers for Irregular Languages.* In Proceedings WCRE 2003, pages 206–215. IEEE Computer Society, November 2003. (p 8)

[LEHM 85] M. M. Lehman and L. Belady. Program Evolution – Processes of Software Change. London Academic Press, 1985. (p 1)

[LESK 75] M. Lesk and E. Schmidt. *Lex — A Lexical Analyzer Generator.* Computer Science Technical Report #39, Bell Laboratories, Murray Hill, NJ, 1975. (p 1)

[MATS 02] Y. Matsumoto. The Ruby Programming Language. Addison Wesley Professional, 2002. (pp 37, 43)

[MOON 01] L. Moonen. *Generating Robust Parsers using Island Grammars.* In Proceedings WCRE 2001, pages 13–22. IEEE Computer Society, October 2001. (pp 2, 8)

[TICH 00] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. *A Meta-model for Language-Independent Refactoring.* In Proceedings of ISPSE ’00 (International Conference on Software Evolution), pages 157–167. IEEE Computer Society Press, 2000. (pp 11, 23)

[VAN 98] M. van den Brand, A. Sellink, and C. Verhoef. *Current Parsing Techniques in Software Renovation Considered Harmful.* In Proceedings of IWPC ’98 (6th International Workshop on Program Comprehension), pages 108–118. IEEE Computer Society, June 1998. (pp 2, 8)