Example-Driven Reconstruction of Software Models

Oscar Nierstrasz, Markus Kobel, Tudor Gîrba
Software Composition Group
Institute of Computer Science
University of Bern
www.iam.unibe.ch/∼scg

Michele Lanza
Faculty of Informatics
University of Lugano
www.inf.unisi.ch

Horst Bunke
Computer Vision and Artificial Intelligence Group
Institute of Computer Science
University of Bern
www.iam.unibe.ch/∼fki

Abstract

As software systems evolve, they become more complex and harder to understand and maintain. Certain reverse engineering techniques attempt to reconstruct software models from source code with the help of a parser for the source language. Unfortunately a great deal of effort may be required to build a specialized parser for a legacy programming language or dialect. On the other hand, (i) we typically do not need a complete parser that recognizes all language constructs, and (ii) we have a rich supply of (legacy) examples. We present an approach to use these facts to rapidly and incrementally develop parsers as follows: we specify mappings from source code examples to model elements; we use the mappings to generate a parser; we parse as much code as we can; we use the exceptional cases to develop new example mappings; and we iterate. Experiments with Java and Ruby, two very different languages, suggest that our approach can be a very efficient and effective way to rapidly construct software models from legacy code.

Keywords: parsing, grammars, reverse engineering

1 Introduction

As software evolves, it becomes more complex and harder to maintain [20]. Additional effort is therefore required to simplify the software. *Reverse engineering* is the process of analyzing a software system to build a higher-level model of that system [5]. Reverse engineering is part of a broader reengineering lifecycle in which software systems are analyzed, models are built, problems are detected, and various measures are taken to rejuvenate and simplify the software to enable further change [7, 24].

Many different sources of information can be exploited to reverse engineer a software system, such as stakeholders' experiences, documentation, bug reports, the running system and so on, but undoubtedly one of the most reliable resources, and sometimes the only one, is the source code itself. Experience shows that in many legacy projects high-level design documentation will be out of sync with the source code, so the high-level models will need to be reconstructed from the code. Various reverse engineering tools and approaches do not operate directly on the source code, but rather parse the code and build an abstract model of the code conforming to some reverse-engineering meta-model. This model is then used as the basis for various analyses, queries and manipulations [25].

A special parser is needed that can construct model elements that are understood by reverse engineering tools. If one is lucky enough to have available a general parsing framework for the language in question, a specialized model builder can be built by a talented and experienced developer with a few days of effort. Unfortunately there exist thousands of programming languages, and even mainstream languages exist in many dialects. For many languages, off-the-shelf parsers that can be adapted to the task of model reconstruction simply do not exist. This simple fact can greatly increase the effort required to build the parser from days to weeks (or worse). As a consequence, a reverse engineering project for a "new" language can be stymied at its very inception due the lack of a suitable parser.

We envision an approach in which a software reverse engineering expert should be able to spend no more than a few hours to construct a model-building parser for source code written in an arbitrary programming language. We exploit two important facts concerning the task at hand:

1. A complete parser is typically not required, as in many analyses, many programming features can be ignored. For example, many useful analyses can already be performed with coarse-grained information about classes, methods, the inheritance relations between classes, and the lines of code inside each method [16].
2. The code of the legacy system offers a large amount of code examples that the reverse engineer understands.

Based on these two facts, we propose the following approach in which:

1. the engineer specifies a set of *mappings* from source code examples to model elements of the reverse-engineering meta-model,
2. the mappings are used to automatically generate a parser for the examples that will directly produce model elements,
3. the resulting parser is applied to some portion of the code base,
4. software artifacts that cannot be parsed are flagged and are used to construct new mapping examples,
5. a new parser is generated and applied to the remaining code base,
6. the process is repeated until all (or enough of) the code is analyzed.

The key benefits of this approach are:

- The engineer specifies mappings, not grammar rules, so does not need to be an expert in parser technology.
- The model-building parser is developed quickly and iteratively. One can interrupt the process when "enough" code has been converted to the the reverse-engineering meta-model. "Enough" in this case is dependent on the reverse engineering goal.
- A single, consistent grammar is not needed. Multiple parsers based on different sets of examples can be used to parse the code base with different strategies.

Structure of the paper. In Section 2 we describe the technical details of our approach. In Section 3 we present an overview of CODESNOOPER, our experimental implementation of *example-driven model reconstruction*. We use CODESNOOPER to reconstruct software models that conform to the FAMIX reverse-engineering meta-model. In Section 4 we discuss two case studies in which CODESNOOPER is applied to Java and Ruby code. We continue in Section 5 with a discussion of the achievements and the current limitations of our approach. We briefly discuss related work in Section 6, and we conclude with some remarks about future and ongoing work in Section 7.

2 Example-Driven Parsing

The initial phases of a reengineering project can be critical for assessing the state of the software and for establishing confidence with various stakeholders [7]. A large number of reverse engineering and software analysis tools and techniques have been developed over the years, but many of them require the source code to be parsed before the analysis technique can be applied. This can be a major obstacle for the vast majority of software projects written in programming languages for which off-the-shelf parsers ready to be adapted to new tasks simply do not exist.

We envision a scenario in which a reverse engineer could quickly develop an *ad hoc* parser generated from examples of mappings from code to model elements. For the approach to be of practical value, the following points should be addressed:

- There should be few assumptions about the host language.
- There should be a simple, high-level interface for specifying mappings from example code to model elements.
- The approach should be able to handle any kind of input, *i.e.*, even code containing errors.
- The approach should be incremental and iterative — specify mappings, generate parser, parse code, identify any code that cannot be parsed, and so on.
- Whenever the generated parsers cannot parse some given code, focussed feedback should be generated indicating which examples could not be parsed, to aid the user in specifying new mappings for those examples.

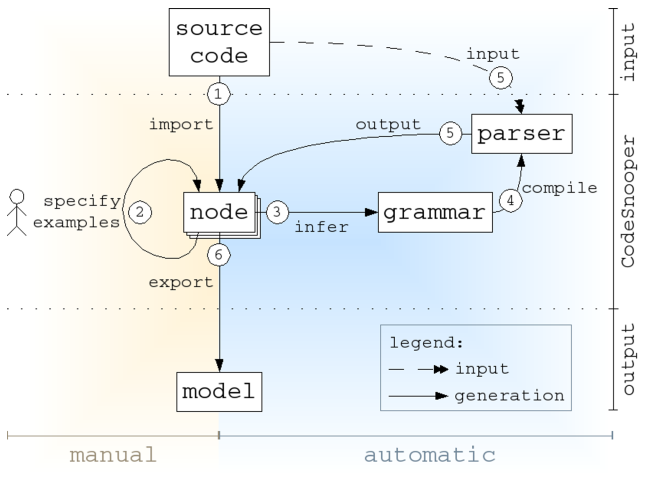


Figure 1. CodeSnooper overview

Figure 1 presents an overview of a typical usage scenario of our approach supported by CODESNOOPER, a proof-of-concept tool to support example-driven model reconstruction (see Section 3).

1. Legacy source code is imported as source code nodes.
2. The user specifies example mappings from code to model elements.
3. The mappings are used to generate a BNF grammar.
4. The grammar is used to generate a parser.
5. Source code is parsed by one or more parsers to produce source code nodes.
6. The parsed nodes are exported as model elements.

2.1 Scanning

Lexical analysis is performed by a simple, generic scanner that breaks source code files into streams of tokens representing identifiers, numbers, comments and whitespace. The parser, rather than the scanner, is given the task of distinguishing which "identifiers" actually represent keywords in the language. Similarly, special character sequences can be recognized directly in the parser as language construct non-terminals.

The effect of this simple approach is that at most a few minutes are dedicated to adapting the scanner to a new language. The reverse engineer can then focus his or her attention on the modeling task, rather than fiddling with the scanner definition. In general, we expect that a set of standard scanner definitions will suffice for most languages, so even this task could be streamlined.

The example depicted below 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]+               ;
```

Figure 2. Java scanner

2.2 Mapping Code to Model Elements

Consider the following snippet of Java code.

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

Our simple scanner will convert this to a stream of tokens. We must now specify how these tokens map to the target model elements of the reverse-engineering meta-model (*e.g.*, class, method — see Section 3). First, we must specify a signature for a class by reducing the example to:

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

The code within the curly brackets (described as <not_known>) does not matter at this stage, since we want to concentrate on the definition of class entity itself.

We must also specify that `class`, `{` and `}` are "keywords" and that `AJavaClass` is the name of the entity. We can similarly specify that a method has the following signature:

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

In this case `public`, `void`, `{`, `}`, `(` and `)` are the keywords and `hello` is the name.

Figure 3 shows how the entire example is represented as a tree of nodes. Each nodes knows which target element of the model it represents, if any. For example, the first node is a Class and has as name `AJavaClass`.

2.3 Grammar generation

Based on the signatures we have specified we can generate a grammar. We traverse the example tree and generate a grammar production for each node that maps to a target element of the meta-model, in our case FAMIX[8]. From the above example we obtain the following grammar rules:

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

The non-terminal of the rule is named after the target element of the node. The production is generated from the signature. Keywords become literals. Subnodes translate to non-terminals unless they do not have a target, in which case they translate to the catch-all target `not_known`.

Suppose we start with a different example in which we are only interested in public methods:

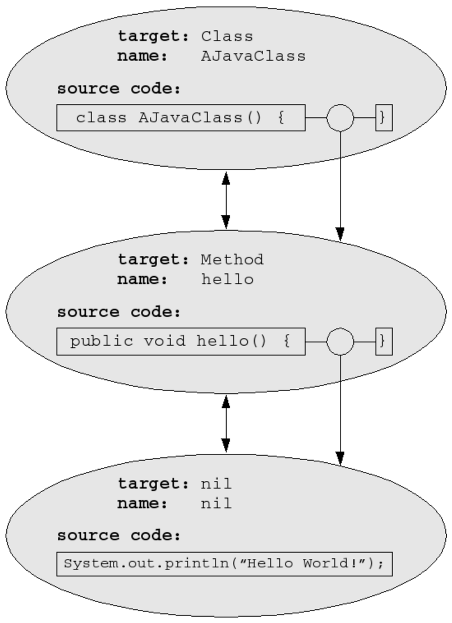


Figure 3. Example nodes

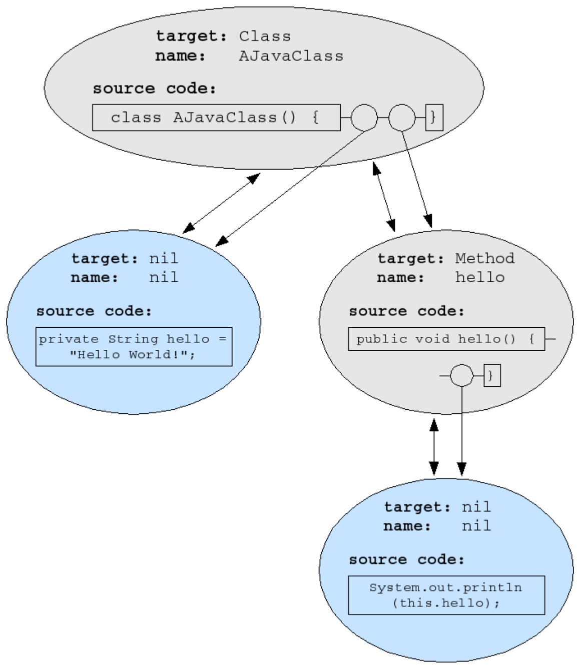


Figure 4. Nodes without a target

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

In this case the nodes representing the private attribute and the body of the public method will have no target, as shown in Figure 4.

This example generates a different grammar than the previous one:

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

The grammars we built are based on multiple examples. This will result in multiple production rules for the same non-terminals, which we have to merge. For example, these rules:

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

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

will be merged into the following grammar:

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

In most cases merging is the right thing to do, but in certain obscure situations this may result in a grammar that accepts invalid code [12]. Since we assume that the legacy code we are parsing is syntactically correct, this is not an issue in practice.

2.4 The generated parser

From the generated grammar we can now generate a parser which will build model elements from the parsed code. If the parser generation fails, we may have to review the examples, regenerate the grammar and attempt to build the parser afresh. The most common difficulty is that the generated grammars may be ambiguous. We will revisit this issue in Section 5, from the point of view of the experiments we conducted on Java and Ruby case studies.

The generated parser builds a parse tree of nodes representing entities of the meta-model. The parse tree can then be processed in a variety of ways, for instance the tree can be traversed by a fixed tool that generates a model description in some interchange format such as XMI[1] or GXL[2].

Note than we can generate more than one parser, and that the multiple parsers can work in parallel. For example, we may specify example mappings for both Java classes and Java interfaces. Instead of being required to generate a single, consistent grammar that will handle both classes and interfaces, we can generate two grammars and two parsers. When one parser fails to recognize a model element, we can simply try another. This leads to a larger number of simpler parsers and somewhat alleviates the problem of dealing with ambiguous grammars.

3 CodeSnooper

CODESNOOPER is a proof-of-concept prototype (see Figure 5) that uses example-driven model reconstruction [12]. CODESNOOPER is implemented in VISUALWORKSSMALLTALK[3] using the SMACC[4] compiler compiler.

3.1 Reverse engineering context

In our specific implementation, we use MOOSE as the target reverse engineering platform [25, 9]. MOOSE is a language-independent reengineering environment that provides a variety of common services for reengineering tools including metrics evaluation and visualization, a model repository, and generic GUI support for querying, browsing and grouping. A key bottleneck in applying MOOSE to different legacy projects is generating software models from source code of new languages.

In MOOSE meta-models are implemented as instances of the MOF[5] meta-meta-model, OMG's meta-object facility. In particular, MOOSE implements the FAMIX reverse and re-engineering meta-model [29, 28] and, as such, FAMIX is an instance of MOF. This allowed for a generic implementation of CODESNOOPER: we did not hardcode CODESNOOPER to work directly with FAMIX, but we actually used the MOF descriptions to generate the mappings from the code to the meta-model elements.

FAMIX is a language-independent meta-model to support reverse engineering and reengineering operations. In Figure 6 we see the core of the FAMIX meta-model. FAMIX is an extensible meta-model which can be adapted to different programming languages and to the needs of different kinds of reverse and reengineering tools. For the purpose of this paper we only focus on the FAMIX core model elements.

[1]http://www.omg.org/technology/documents/formal/xmi.htm
[2]http://www.gupro.de/GXL/
[3]http://smalltalk.cincom.com
[4]http://www.refactory.com/Software/SmaCC
[5]http://www.omg.org/mof/

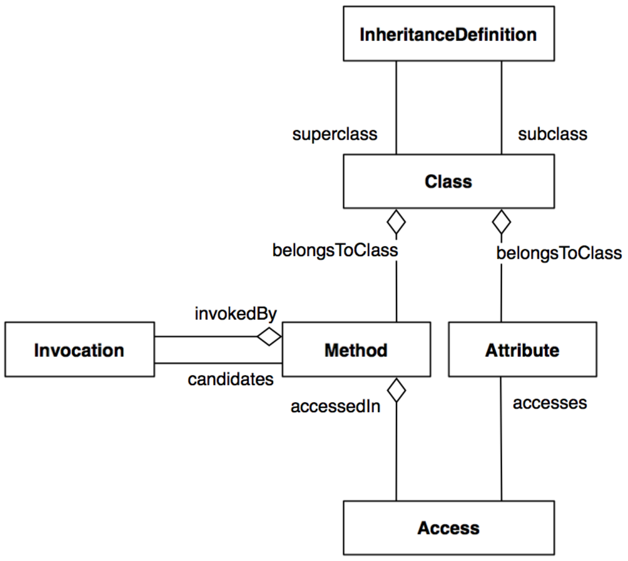


Figure 6. Famix

We have used example-driven model reconstruction as a front-end to build FAMIX source models for MOOSE.

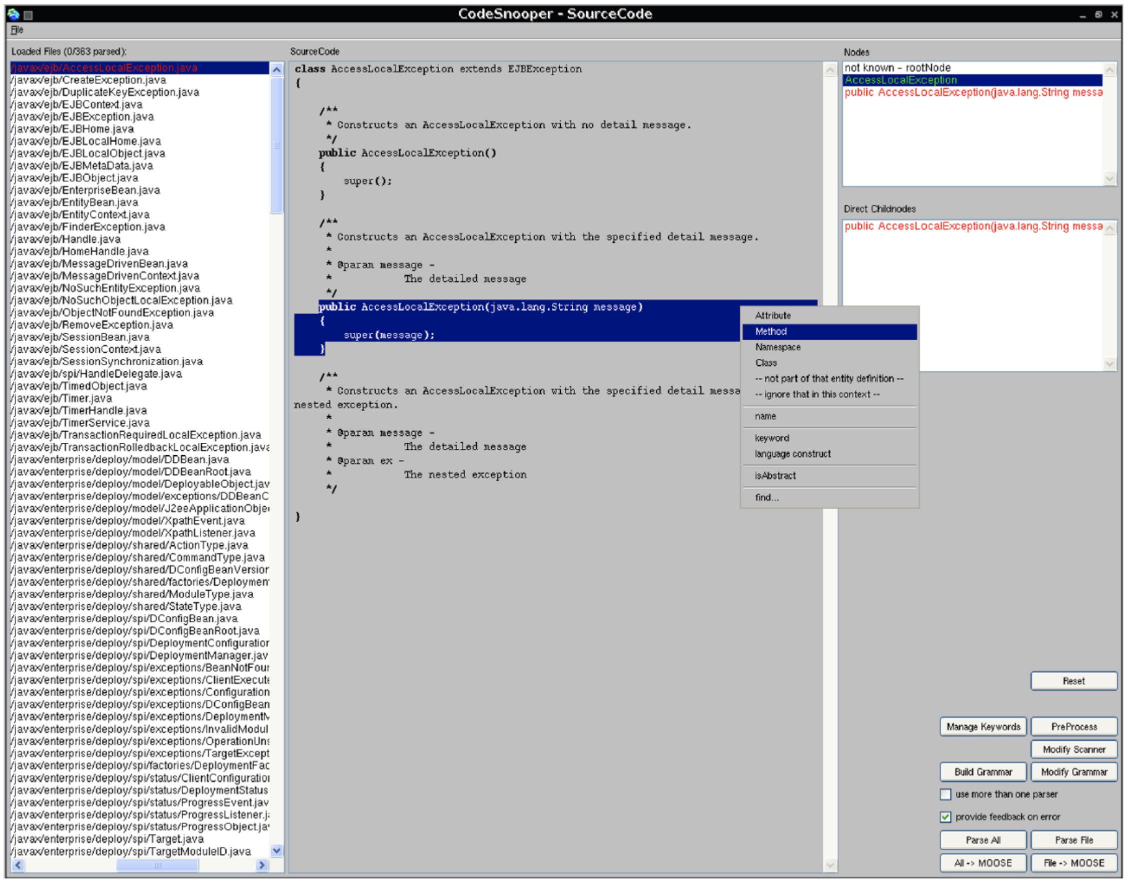
3.2 Generating the model

A scanner definition must be provided for each language, but this is usually a straightforward task. CODESNOOPER focuses instead on the interface needed for dynamically specifying the mapping from code examples to model elements.

In Figure 5 we see a portion of CODESNOOPER's user interface which allows one to select syntactic elements and flag them as representing either certain language constructs (*i.e.*, "keywords") or FAMIX model elements. CODESNOOPER also provides means to manage keywords, generate a grammar, modify the grammar or the scanner, and to parse the set of input files with one or more generated parsers. The file list (Figure 5, left-hand pane) is updated to indicate which source files have been successfully parsed or not.

4 Case studies

We have applied CODESNOOPER to two very different case studies to assess the feasibility of example-driven model reconstruction. The Java case study allows us to assess the recall achieved with CODESNOOPER by comparison with results obtained with a robust Java parser that is available to us for loading software models into MOOSE from Java source code. The Ruby case study allows us to



```
/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/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
```

```java
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
     */

}
```

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

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

Figure 5. CodeSnooper: Main View while specifying an example

assess the approach when applied to a language with a syntax that is very different from Java.

4.1 JBoss

As first case study we worked with JBoss, an open source implementation of the J2EE application server specifications. We analyzed the jboss-j2ee package of JBoss, consisting of 363 Java files.

We proceeded iteratively, starting with just three examples:

1. The first example is a normal Java class that we map to a FAMIX Class without any attributes set. `javax/ejb/AccessLocalException.java`
2. The second example is an abstract Java class that we map to a FAMIX Class. For that entity we also set the “isAbstract” attribute to true. `javax/xml/soap/SOAPPart.java`
3. As third example we take a Java interface that we map also to a FAMIX Class. We also set the ‘isAbstract’ attribute for that entity to true. `javax/ejb/EJBObject.java`

This yields the following grammar (reduce actions not shown):

```
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* "}" ;
```

The resulting parser can parse 355 of the 363 Java files. Of the eight files that cannot be parsed, two contain unbalanced brackets or comment characters within a string and the other six use the keyword `class` in an unexpected context (*i.e.* to denote an inner class). To solve the first problem we would just need a slightly more sophisticated scanner to remove the problematic strings.

We can solve the second problem by ignoring `class` in the context of a class. In this way we can parse more files, but we fail then to detect inner classes. Comparing this first result to that obtained with a robust parser we find that we only miss five classes:

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

In a second iteration we give examples of methods in abstract and concrete classes as well as interfaces. This leads to three separate grammars which cannot easily be merged since this would lead to an ambiguous grammar [12]. Instead we generate three parsers and apply them in parallel to the source files. We now obtain the following results:

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

In addition to the two files we could not parse earlier, we now have some problems due to (i) attributes being confused with methods, (ii) language constructs (like `static`) occurring in unexpected contexts, (iii) different kinds of definitions of methods. Additional examples would help to solve these problems.

In a third iteration we add examples to recognize attributes. Once again we obtain three parsers based on three sets of examples for abstract classes, concrete classes and interfaces. We obtain the following results:

	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

This process can be repeated to cover more and more of the subject language. The question on when to stop can be answered with "When the results are good enough". Good enough in this context means when we have enough information for a specific reverse engineering task. For example, a "System Complexity View" [18] is a visualization used to obtain an initial impression of a legacy software system. To generate such a view we need to parse a significant number of the classes, identify subclass relations, and establish the numbers of methods and attributes of each class. Even if we parse only 80% of the code, we can still get an initial impression of the state of the system. If on the other we would want to display a "Class Blueprint" [17], a semantically enriched visualization of the internal structure of classes we would need a refined grammar to extract more information. The "good enough" is thus given by the reverse engineering goals, which vary from case to case.

4.2 Ruby

As second case study we chose the language Ruby, because it is quite different from Java and it has a non-trivial grammar. We took the unit testing library distributed with Ruby version 1.8.2 released at the end of 2004. This part of the library contains 22 files written in Ruby. We do not have a precise parser for Ruby that can generate a FAMIX model (actually, to our knowledge, for Ruby there is only one precise parser, namely the Ruby interpreter itself). Instead we retrieve the reference model by inspecting the source code manually.

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. *Modules* also have the function of *Namespaces*. Ruby does not support *Abstract Classes* [22].

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>        ;
```

Using just 2 examples each of namespaces, classes, methods and attributes, we are able to parse 7 of the 22 files.

	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

Amongst the files we could not parse, there are 4 large files containing GUI code. If we ignore these files, we are able to detect about 25% of the target elements.

There are two main reasons that so few files can be successfully parsed:

1. The comment character # occurs frequently in strings and regular expressions, causing our simple-minded scanner to fail. A better scanner would fix this problem. With some simple preprocessing (removing any hash character that occurs inside a string and removing all comments) we can improve recall to 65-85%.

2. Ruby offers a very rich syntax for control constructs, allowing the same keywords to occur in many different positions and contexts. One would need many more examples to recognize these constructs.

5 Discussion

Our experience with these preliminary case studies demonstrates that the idea of example-driven model reconstruction is feasible: using only a naive scanner and a few examples that map source code to model elements we can generate parsers that build models of a significant portion of the total source code to models. In the ideal case, the user must only invest a modest amount of time (*i.e.*, hours rather than days) to reconstruct a usable software model. Our observation is that the 80/20 rule applies in this case: it is straight forward to parse a relevant amount of code (say 80%), but very time-consuming to generate a full parser, which is however not needed in most cases. The end of the reconstruction process is thus given by the reverse engineering context, *i.e.* as soon as we can parse enough code to allow us to perform a specific type of analysis, we stop the reconstruction process and concentrate on the analysis.

To make the approach really practical for a realistic range of languages, and robust enough for users not expert in parser technology, however, a number of issues need to be resolved, and more extensive case studies need to be carried out. Although there are numerous shortcomings and obstacles, the path towards a practical and usable approach is relatively clear.

The first problem is that of *ambiguous grammars*. Although the parser generator we used (SmaCC) can deal with ambiguous grammars, the results are often not usable since conflicts may not be resolved. As a consequence CODESNOOPER rejects examples that lead to ambiguous grammars. When this occurs, the user could either try to specify different examples, or use multiple sets of examples to generate multiple, unambiguous grammars, for multiple parallel parsers.

Another possibility to cope with ambiguity is probabilistic grammars [32]. Here a probability is assigned to each grammar production. Consequently the probability of a parse tree (or, equivalently, an input string) can be computed and in case a code fragment has several possible derivations, a decision can be made for the one that has the highest probability.

The next difficulty concerns *false positives* (code fragments classified as the wrong kind of model element) and *false negatives* (missed model elements). As long as no parse errors occur and no robust parser is available, false positives or negatives can only be detected manually. More examples are needed to generate more precise parsers. It is possible that a more sophisticated user interface could help by allowing the user to mark which source code has been correctly parsed. Code already correctly classified could then be used as a benchmark to test the quality of parsers generated from newer examples, thus compensating partially for the lack of a robust parser.

Incorrectly identified tokens are a major source of problems. Since we are using very simple-minded scanners, tokens like `class` may be recognized as keywords when they should not be. There are essentially two solutions: either we can ignore more things — *i.e.*, we ignore certain keywords like `class` within a given context, thus possibly losing information — or we can detect more things — *i.e.*, we can work with richer scanners. It is easy to imagine that a small library of moderately rich scanners could be used to cover a wide range of programming languages. However a key assumption of the entire approach is that the reverse engineer should not be required to directly edit either the scanner or the parser specifications. An open research question is whether the approach could be generalized to "example driven *scanning*".

Complex control structures in languages like Ruby are another thorny issue. Keywords like `end` are used both to delimit the model elements we are looking for as well as within expressions that occur inside those elements. To correctly recognize methods and classes, we must also recognize constructs that we may not necessarily be interested in. Once again additional examples may be needed to correctly identify the boundaries of model elements.

6 Related work

Many reengineering frameworks use a form of *fuzzy parsing* in order to support more programming languages or more dialects of the same programming language. A fuzzy parser extracts a partial source code model by skipping all input until an *anchor terminal* is found. Then usual context-free analysis is attempted using a production starting with the found anchor terminal [13].

With island grammars [23] we get robust 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). Our approach exploits this idea since our mappings may specify that certain constructs correspond to "water" (*i.e.*, the `not_known` parts). By varying the amount and details in productions for the constructs of interest, we can trade off accuracy, completeness and development speed. Useful variants of island grammars include *lake grammars* (a grammar extended with productions for water), *island with lakes* and even *lakes with islands* [23].

In contrast to island grammars, a tolerant grammar [11] uses an already available single base-line grammar as a point-of-reference to reduce both false positives and false negatives.

The problem we address in this paper is related to grammatical inference [6]. Given a set S of sentences, the task of grammatical inference is to derive a grammar that generates S. This task, however, is more general than the problem considered in this paper as we assume that a mapping from the examples to the target model elements is known.

RegReg is a generator of robust parsers for irregular languages [19]. 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. RegReg can be used to implement both island and fuzzy parsing.

A generalized LR parser (GLR) uses parallel parsers to explore different ways to proceed when shift/reduce or reduce/reduce conflicts arise. If the conflict is due to the need for a lookahead, the forked parsers die. 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. The number of trees is reduced by applying syntactic disambiguation rules. If there is more than one tree left over at 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 [31].

DURA is a parser generator that uses backtracking to resolve conflicts. Compared with a plain LR parser, DURA-generated parsers provide an "undo" operation in addition to "shift" and "reduce". DURA takes a more optimistic view than GLR parsing: Not only can a plain LR parser handle most of the input, but in case of conflicts, it does not need to go very far to backtrack if it selects the wrong path [4].

Earley parsing [10, 3] is a technique that can parse any context-free grammars. It can cope in particular with ambiguous grammars. There has been renewed interest in Earley parsing for implementing little languages [2] and domain-specific languages [30]. A powerful extension of Earley parsing is minimum distance error-correcting parsing [1, 21]. This technique is based on error productions that can be automatically generated and are added to a grammar. A minimum distance error-correcting parser is able to handle any arbitrary syntactically ill-formed input string, x, by providing the most similar element, y, from the underlying language, together with the parse trees of x and y.

Revealer [26, 27] is a reverse engineering tool that uses a pattern language to recognize architectural elements in source code. Patterns are specified as XML documents conforming to the Revealer DTD. Revealer combines lexical and syntactic analysis by allowing the user to specify just the code fragments of interest.

Lämmel and Verhoef have developed an approach to semi-automatically recover grammars of legacy languages from numerous resources, including language references, compilers and other artifacts [15, 14]. They have been able, for example, to construct a running parser for VS COBOL II in a few weeks for use in a variety of tools, considerably less than the 2-3 man-years estimated. They use a series of techniques to automatically extract grammars from various sources, automate testing of parsers, and transform grammars specified with diverse formalisms.

The key idea behind this paper is to to use mappings from example source code fragments to model elements to automatically generate parsers that will recognize model elements.

7 Conclusions

Example-driven model reconstruction offers a lightweight means to quickly construct software models for legacy software in the absence of a specialized robust parser for the programming language in question. In principle, a reverse engineer could spend a few hours developing examples to load the software model, and spend more productive time analyzing the model.

CODESNOOPER, our proof-of-concept prototype demonstrates that the process of specifying mapping examples and generating parsers can be driven by a simple graphical user interface. Case studies have shown that by specifying only a few examples, one can automatically generate software models that cover a large portion of Java code.

More difficulties were encountered with Ruby code, for technical reasons that can surely be surmounted. The parsers generated from the example mappings are often ambiguous. This problem is alleviated by the fact that we can generate multiple unambiguous parsers instead, and apply them in parallel to the source code. Another possibility to deal with ambiguous grammars is to generate an Earley parser instead of an LALR parser.

The experiments only made use of minimal scanners. It is likely that more sophisticated scanners would improve the quality of the resulting parsers. This however would place more of a burden on the end user. It is possible that libraries of relatively standard scanners for comments, strings and other most common constructs would reduce this burden.

Acknowledgments. We thank the anonymous referees who suggested many improvements. We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects "A Unified Approach to Composition and Extensibility" (SNF Project 200020-105091/1) and "NOREX — Network of Reengineering Expertise" (SNF Project IB7320-110997).

References

[1] A. V. Aho and T. G. Peterson. A minimum distance error-correcting parser for context-free languages. *SIAM Journal of Computing*, 1:305–312, 1972.

[2] J. Aycock. Compiling little languages in Python. In *Proc. 7th Int. Python Conf.*, pages 69–77, Nov. 1998.

[3] J. Aycock and N. Horspool. Practical Earley parsing. *The Computer Journal*, 45(6):620–630, 2002.

[4] D. Blasband. Parsing in a hostile world. In *Proceedings of the Eight Working Conference on Reverse Engineering (WCRE 2001)*, pages 291–300. IEEE Computer Society, Oct. 2001.

[5] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[6] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332–1348, 2005.

[7] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2002.

[8] S. Demeyer, S. Tichelaar, and S. Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[9] 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, Milano, 2005.

[10] J. Earley. An efficient context-free parsing algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[11] S. Klusener and R. Lämmel. Deriving tolerant grammars from a base-line grammar. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 179–188. IEEE Computer Society, Sept. 2003.

[12] M. Kobel. Parsing by example. Diploma thesis, University of Bern, Apr. 2005.

[13] R. Koppler. A systematic approach to fuzzy parsing. *Software: Practice and Experience*, 27(6):637–649, 1996.

[14] R. Lämmel and C. Verhoef. Cracking the 500-language problem. *IEEE Software*, 18(6):78–88, Nov. 2001.

[15] R. Lämmel and C. Verhoef. Semi-automatic grammar recovery. *Software—Practice & Experience*, 31(15):1395–1438, Dec. 2001.

[16] M. Lanza. *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. PhD thesis, University of Berne, May 2003.

[17] M. Lanza and S. Ducasse. A Categorization of Classes based on the Visualization of their Internal Structure: the Class Blueprint. In *Proceedings of OOPSLA '01 (International Conference on Object-Oriented Programming Systems, Languages and Applications)*, pages 300–311. ACM Press, 2001.

[18] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *IEEE Transactions on Software Engineering*, 29(9):782–795, Sept. 2003.

[19] M. Latendresse. RegReg: a lightweight generator of robust parsers for irregular languages. In *Proceedings Tenth Working Conference on Reverse Engineering (WCRE 2003)*, pages 206–215. IEEE Computer Society, Nov. 2003.

[20] M. Lehman and L. Belady. *Program Evolution: Processes of Software Change*. London Academic Press, London, 1985.

[21] G. Lyon. Syntax-directed least-errors analysis for context-free languages: a practical approach. *Commun. ACM*, 17(1):3–14, 1974.

[22] Y. Matsumoto. *Ruby in a Nutshell*. O'Reilly, 2001.

[23] L. Moonen. Generating robust parsers using island grammars. In E. Burd, P. Aiken, and R. Koschke, editors, *Proceedings Eight Working Conference on Reverse Engineering (WCRE 2001)*, pages 13–22. IEEE Computer Society, Oct. 2001.

[24] O. Nierstrasz, S. Ducasse, and S. Demeyer. Object-oriented reengineering patterns — an overview. In M. L. Robert Glück, editor, *Proceedings of Generative Programming and Component Engineering (GPCE 2005)*, pages 1–9. LNCS 3676, 2005. Invited paper.

[25] O. Nierstrasz, S. Ducasse, and T. Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE 2005)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[26] M. Pinzger, M. Fischer, H. Gall, and M. Jazayeri. Revealer: A lexical pattern matcher for architecture recovery. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE 2002)*, pages 170–178, 2002.

[27] M. Pinzger and H. Gall. Pattern-supported architecture recovery. In *10th International Workshop on Program Comprehension (IWPC'02)*, pages 53–61, 2002.

[28] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Berne, Dec. 2001.

[29] S. Tichelaar, S. Ducasse, S. Demeyer, and O. Nierstrasz. A meta-model for language-independent refactoring. In *Proceedings of ISPSE '00 (International Symposium on Principles of Software Evolution)*, pages 157–167. IEEE Computer Society Press, 2000.

[30] L. Tratt. The Converge programming language. Technical Report TR-05-01, Department of Computer Science, King's College London, Feb. 2005.

[31] M. van den Brand, A. Sellink, and C. Verhoef. Current parsing techniques in software renovation considered harmful. In S. Tilley and G. Visaggio, editors, *Proceedings of 6th International Workshop on Program Comprehension (IWPC '98)*, pages 108–117. IEEE Computer Society, June 1998.

[32] C. S. Wetherell. Probabilistic languages: A review and some open questions. *ACM Computing Surveys*, 12(4):361–379, 1980.