Generating logic-based representations for programs

Călin Jebelean
Politehnica University of Timişoara
Computer Science Department
Bd. V. Pârvan 4, Timişoara
calin.jebelean@cs.upt.ro

Ciprian-Bogdan Chirilă
Politehnica University of Timişoara
Computer Science Department
Bd. V. Pârvan 4, Timişoara
ciprian.chirila@cs.upt.ro

Anca Măduţa
Politehnica University of Timişoara
Computer Science Department
Bd. V. Pârvan 4, Timişoara
ancamaduta@waterford.org

Abstract

Modern software engineering has come to a point where it deals with quite large and complex software artifacts. Labor-intensive activities such as code analysis and code transformation are becoming less and less tractable on such enormous software systems unless a certain level of automatization is provided. Since automatic approaches of code analysis and code transformation strongly rely on software models instead of actual software systems, the software modeling process is of vital interest to a great deal of researchers in the software engineering community. However, the main drawback of most of the software modeling tools available is the fact that they are aimed at software systems written in a certain programming language. This article introduces ProGen, a software tool capable of modeling software systems written in any language for which a plain JavaCC *grammar is available, also describing its advantages and limitations.*

1 Introduction

Code analysis ([3], [9], [14], [11], [8]) and code transformation ([10]) are two activities of great interest for modern software engineers. Code analysis could prove useful in the process of detecting bad programming habits (like memory allocation errors, for example). It can also be used for more important issues (especially in object-oriented software engineering) such as detecting design flaws, places where good design guidelines have been misinterpreted or even ignored. Each result of such an analysis would be a potential candidate for a code transformation, which shows its usefulness in correcting design problems or even bugs in the system.

Both code analysis and code transformation are processes that rely on software models instead of actual software systems. Performing analyses and transformations directly on the source code of a software system is a difficult task and each tool capable of doing this would be almost impossible to reuse for a different programming language than the one it was initially designed for. This is true since such a tool would have to mix parsing actions with activities that perform analyses and transformations on the source code in a monolithic, hardly reusable structure.

On the other hand, a software model contains information about the target software system in a very accessible format (a relational database, for example ([11])). It could also offer easy access to specific relations between software entities that may not be very obvious to a code parser based on the grammar of the programming language used (for example, a relational database could easily link the name of a class in an object-oriented software system to the names of all the methods belonging to that class). However, the main advantage of a software model lies in the fact that it provides an interface to the actual software system. Model generation and model analysis/transformation are different tasks now and their code is no longer mixed. Model generation needs a parser for the specific programming language used and generates a software model that can be interrogated (model analysis) or transformed (model transformation) further (see figure 1).

It should be easier now to reuse analyses and transformations on the software model even if the software artifact is reimplemented using a different programming lan-

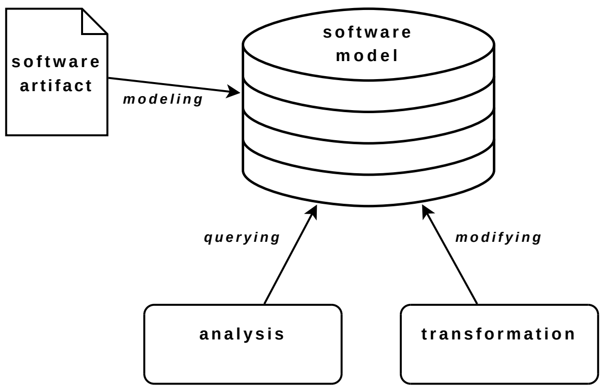

Figure 1. Software models

guage, provided that both languages have similar philosophies (for example, they are both object-oriented). What needs to change is the model generator, the tool that parses the source code of the software artifact and generates the model.

In this paper, we address the problem of generating software models in a language independent fashion. We introduce a tool called PROGEN (PROlog GENerator) that is capable of parsing source code and producing logic models for software systems written in any programming language. For that purpose, PROGEN is equipped with a grammar repository for several widely used programming languages. Thus, the model generation process in PROGEN is grammar-driven and produces PROLOG facts corresponding to each node in the abstract syntax tree representation of the software system. Using this approach, one shouldn't use different model generators for different programming languages, but rather configure the same model generator (PROGEN) with different input grammars ([2]).

The paper is structured as follows. Section 2 presents the logic metamodel we use. Section 3 describes the steps of our approach and offers an introspection in the implementation of PROGEN. Section 4 discusses several issues related to model optimization and potential uses of the output models. Section 5 presents related approaches to model generation and section 6 concludes.

2 The PROLOG metamodel

To achieve the goal of modeling programs written in different programming languages, we have chosen PROLOG (a declarative language) to represent meaningful information about the source code under analysis. Our PROLOG representation is generic and suitable for any programming language or model specified by a context-free grammar.

In order to describe the PROLOG metamodel we use, we will discuss the factbase generation on a simple example. Figure 2 contains a small code snippet written in a demonstrative programming language inspired from the C language specification. It only contains one block of one or more instructions, each instruction is an assignment and expressions can only contain identifiers, constants and the four basic arithmetic operators plus parantheses.

```
{
  b = 4;
  a = ( 8 + b ) * 5;
}
```

Figure 2. A sample program (C subset)

As mentioned earlier, the modeling process is language-independent. This means we do not make any assumption about the specific programming language used in figure 2. Thus, the model generation process is based on the meta-information associated with the code sample, namely on its grammar specification.

The grammar of this demonstrative language is presented in figure 3 using an EBNF notation ([1]). To keep the example simple, we added to the grammar only the necessary rules that would make the code in figure 2 compliant.

```
Block ::= '{' InstrList '}'

InstrList ::= ( Instr ';' )*

Instr ::= Assignment

Assignment ::= <ident> '=' Expr

Expr ::= Term
  ( AdditiveOp Term )*

Term ::= Factor
  ( MultiplicativeOp Factor )*

AdditiveOp ::= '+' | '-'

Factor ::= <ident>
  | <constant>
  | '(' Expr ')'

MultiplicativeOp ::= '*' | '/'
```

Figure 3. A sample grammar (C subset)

The PROLOG metamodel is listed in figure 4. It contains PROLOG clauses derived from the actual grammar rules in figure 3. Each grammar rule has an associated PROLOG clause with the same name in downcase. Furthermore, each clause is associated with an unique integer identifier and also with the identifier of its parent clause, as dictated by the grammar. The only exception to this rule is the first clause, the one associated with the starting non-terminal of the grammar, which has no parent clause.

```
block(#ID).
instrlist(#ID , #blockID).
instr(#ID , #instrlistID).
assignment(#ID , #instrID).
expr(#ID , #assignmentID).
term(#ID , #exprID).
additiveop(#ID , #exprID).
factor(#ID , #termID).
multiplicativeop(#ID , #termID).

atom(#blockID , '{').
atom(#blockID , '}').
atom(#instrlistID , ';').
atom(#assignmentID , '<ident>').
atom(#assignmentID , '=').
atom(#additiveopID , '+').
atom(#additiveopID , '-').
atom(#multiplicativeopID , '*').
atom(#multiplicativeopID , '/').
```

Figure 4. The PROLOG metamodel

For example, each instruction in figure 2 is modeled by an *instr* clause having its own ID as first argument and the ID of its parent as second argument. This parent ID must be the personal ID of an *instrlist* clause since grammar rules (figure 3) specify that each *Instr* is part of an *InstrList*. Atoms in the grammar (terminals) are modeled by using *atom* clauses. An atom is somewhat different than a non-terminal of the grammar in that it doesn't have children so it doesn't need a personal identifier for others to refer to it. Each *atom* clause has the ID of its parent as first argument and its actual value as second argument.

Using instances of clauses shaped by the metamodel in figure 4, the actual PROLOG model of the code snippet in figure 2 is obtained. It is listed in figure 5. Since the factbase representation in figure 5 may not be easily comprehensible, we also provide in figure 6 an equivalent, yet graphical view of the abstract syntax tree (AST) of the program in figure 2. The dotted line follows the AST border and meets the atoms in the actual order they are encountered in the initial source file.

```
block(10000).
atom(10000,'{').
instrlist(10001,10000).
instr(10002,10001).
assignment(10003,10002).
atom(10003,'b').
atom(10003,'=').
expr(10004,10003).
term(10005,10004).
factor(10006,10005).
atom(10006,'4').
atom(10001,';').
instr(10007,10001).
assignment(10008,10007).
atom(10008,'a').
atom(10008,'=').
expr(10009,10008).
term(10010,10009).
factor(10011,10010).
atom(10011,'(').
expr(10012,10011).
term(10013,10012).
factor(10014,10013).
atom(10014,'8').
additiveop(10015,10012).
atom(10015,'+').
term(10016,10012).
factor(10017,10016).
atom(10017,'b').
atom(10011,')').
multiplicativeop(10018,10010).
atom(10018,'*').
factor(10019,10010).
atom(10019,'5').
atom(10001,';').
atom(10000,'}').
```

Figure 5. The PROLOG model

3 Software instrumentation: PROGEN

3.1 Solution overview

To implement the ideas presented in section 2, we have chosen the solution depicted in figure 7.

The main flow of the diagram follows the dotted arrow in figure 7. It starts with an input source file written in some language L and ends with an equivalent PROLOG model for that input file, as described in section 2. In order to achieve this goal, the input file must be fed to a specific parser for language L that generates its logic-based representation.

Such a parser may be manually written or automati-

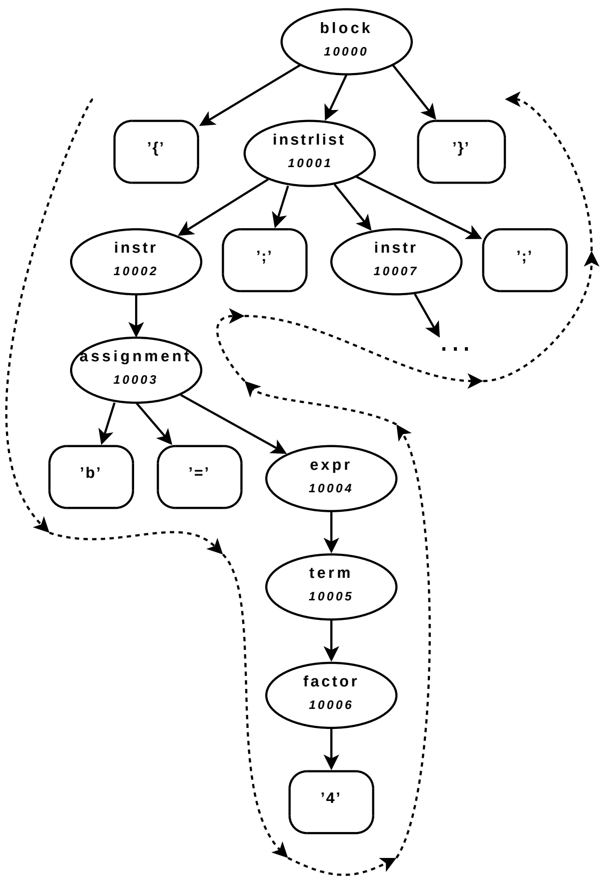

Figure 6. AST representation

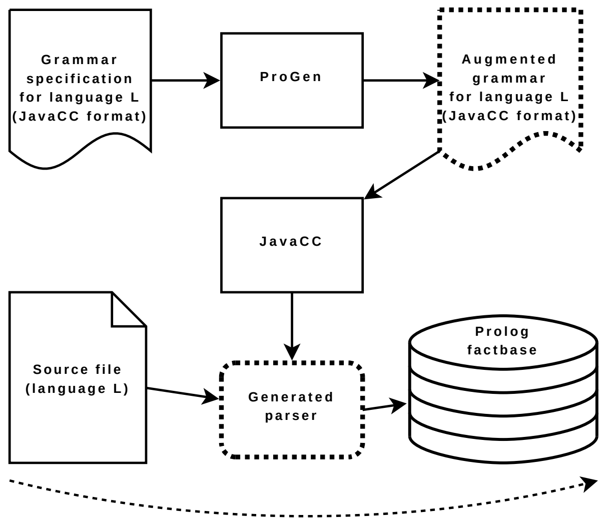
Grammar specification for language L (JavaCC format)
ProGen
Augmented grammar for language L (JavaCC format)
JavaCC
Source file (language L)
Generated parser
Prolog factbase

Figure 7. PROGEN **diagram**

cally generated. A manually written parser is restricted to only one programming language and providing support for another language would involve writing another parser from scratch. An alternative solution would be to devise a methodology which is able to automatically generate such parsers starting from the grammar specification of the programming language of interest. Thus, we are no longer bound to a certain programming language but we can switch easily between different programming languages. This is suggested by the upper blocks in figure 7. We have marked with a dotted border the entities that are the result of automatic generation.

1. Starting from the grammar specification for our concrete language L in JAVACC format, we use PROGEN to generate a syntax-directed translation scheme, which is the very same grammar augmented with generative semantic actions (the format for the augmented grammar is also compatible with JAVACC). This process will be discussed next.

2. The second step implies running JAVACC on the augmented grammar and obtaining a parser for language L that is also capable of generating logic facts because of the semantic actions added earlier.

The reasons why we used JAVACC ([13]) representations for our input grammars are many:

- JAVACC is a powerful parser generator that is able to produce JAVA parsers for virtually any imaginable language;

- the grammar format accepted by JAVACC is quite simple and easy to understand and use;

- JAVACC has a powerful mechanism of grammar augmentation with semantic actions and we may easily use this mechanism to provide syntax-directed fact-base generation during parsing;

- last but not least, JAVACC provides a large online repository of JAVACC-compliant grammars for some of the most well-known programming languages, so it is trivial to also endow PROGEN with a rich grammar repository, making the model generation process possible for programs written in any of these languages.

3.2 Implementation details

We will present in greater detail the first step of the process which involves the tool we propose (PROGEN), since the second step is quite straightforward, JAVACC being a very mature and widely used parser generator. PROGEN's purpose is to take an input grammar for some programming language L in JAVACC format and augment it. As result, we end up with the same grammar annotated with syntax-directed semantic actions that will be responsible for the fact generation process during parsing. The new, annotated grammar also conforms to the JAVACC format so we can use JAVACC during the second step to generate a parser for the programming language L based on this grammar. Once this parser runs on a program P written in L, it will output the PROLOG representation of P (see figure 7).

In order to properly understand the grammar augmentation process, we present the grammar rule that describes an *Assignment* in JAVACC format and augment it:

```
void Assignment() :
{
 /* declarations and initializations
}
{
 <ident>
 "="
 Expr()
}
```

Figure 8. The Assignment grammar rule

```
void Assignment(int parentID) :
{
 /* declarations and initializations
 Token t;
 int id = generateUniqueID();
 gen("assignment(<id>,<parentID>).");
}
{
 t = <ident>
   {
    gen("atom(<id>,'<t>').");
   }

 "="
   {
    gen("atom(<id>,'=').");
   }

 Expr(id)
}
```

Figure 9. The augmented Assignment grammar rule

Further explanations are required for figure 9:

- *generateUniqueID()* is a function that returns a unique integer identifier when called. Practically, this is the function responsible for generating the numeric identifiers from figure 4 and figure 5;
- *gen(String s)* is a function that outputs *s* on the screen or in a file;
- The <*V*> notation (where *V* is a variable) means *the value of V*.

The grammar-annotation process is governed by the following rules:

- each grammar non-terminal NT should receive an integer parameter called *parentID* that represents the numeric identifier of NT's parent in the abstract syntax tree; the first grammar non-terminal is an exception to this rule since it has no parent, so it will receive no parameter;
- each grammar non-terminal NT should call *generateUniqueID()* in its *declarations and initializations* section to obtain a unique integer identifier that will represent NT further;
- each grammar non-terminal NT should call *gen()* in its *declarations and initializations* section to produce a Prolog fact having the same name as NT (only in downcase) and two parameters: the numeric identifier of NT (id, generated earlier) and the identifier of NT's parent (parentID, received as parameter);
- each terminal T in the body of a grammar non-terminal NT should be associated with a call to *gen()* which produces a corresponding *atom* PROLOG fact;
- each non-terminal NT' in the body of a grammar non-terminal NT should receive the numeric identifier of NT as parameter (id);

The semantic actions in figure 9 provide the generated parser (see figure 7) with factbase generation capabilities. For example, each time an assignment is encountered by the parser, the *Assignment* syntactic function of the parser is called and an *assignment* clause is generated in the output file because of the first call to *gen()* in figure 9. It is easily observable in figure 5 that there are exactly two *assignment* clauses, corresponding to the two assignments in figure 2.

As one may notice, the augmentation of the *Assignment* grammar rule (the transformation from figure 8 to figure 9) is a mechanic process and we managed to automate it for any input grammar that conforms to the JAVACC format.

4 Further discussions

4.1 Abstract syntax tree navigation

The PROLOG models produced by our approach are basically PROLOG representations of abstract syntax trees for different kinds of programs. The numeric identifiers associated with each PROLOG clause are nothing more than simple means to represent father-son relations in these trees. Once the PROLOG representation is obtained for a program, it is quite easy to write PROLOG rules to navigate within this abstract syntax tree.

4.2 Model advantages and limitations

When dealing with programs written in a specific, widely used language (such as JAVA or C++), our model generator is no match for the existing model generators that are directly aimed at parsing these languages. While we only produce a simple abstract syntax tree representation, a JAVA model generator, for example, could provide the output model with direct relations between major entities that populate a JAVA system's universe. A PROLOG clause that models the name of a class in a JAVA system could also refer to a list of PROLOG clauses that model the methods belonging to that class. Each clause that models a method could also be linked to a list of clauses modeling parameters of that method, or exceptions that the method potentially raises, and so on.

However, the main advantage of our approach is its language independency. We are able to generate PROLOG representations for any program, written in any language, as long as we have a JAVACC grammar for that language. This is where the JAVACC grammar repository comes in handy. If a model generator for a certain programming language is needed and none exists, a software engineer would have two options: to manually write such a generator or to use the PROGEN approach and automatically generate one and adapt its output to his/her needs later. Unfortunately, PROLOG experience is needed to do that, but this drawback is not quite painful since PROLOG is a declarative language rather than an imperative one, which makes it quite easy to learn and use.

4.3 Chain shortcutting

One important issue that needs further discussion refers to the PROLOG chains of clauses that a grammar-driven approach produces. The output in figure 5 contains many examples of clause chains, but we will exemplify by taking only one of them:

The part of the model displayed in figure 10 describes the assignment *b = 4* from figure 2. Even if the expression assigned to *b* is very simple (a single atom), our syntax-directed approach modeled every grammar non-terminal encountered and produced a clause chain: *expr(10004, 10003), term(10005, 10004), factor(10006, 10005)* and *atom(10006, '4')* because factors compose terms and terms compose expressions, according to the grammar. The same

```
assignment(10003,10002).
atom(10003,'b').
atom(10003,'=').
expr(10004,10003).
term(10005,10004).
factor(10006,10005).
atom(10006,'4').
```

Figure 10. A PROLOG clause chain

clause chain is visible in figure 6. It is very interesting to study how these chains can be eliminated, or at least shortened, because the inner clauses don't provide essential information to the model user. In the example mentioned here, only the initial *expr* clause and the final *atom* clause are relevant to the user.

5 Related works

The main work that inspired PROGEN is the JTRANSFORMER framework presented in [6]. JTRANSFORMER is a complete model generator for JAVA implemented as an Eclipse ([4]) plug-in and it produces PROLOG representations for software systems written in this particular programming language. However, the output model is not syntax-driven. It directly encompasses complex relations between program entities, so that the model user will have easier ways of referring to them. For example, a class modeled by JTRANSFORMER will also contain references to all the class members (attributes and methods) by means of their numeric identifiers. This is possible in JTRANSFORMER because it is aimed at software systems written in JAVA while PROGEN is unable to infer relations other than the simple relation between a grammar non-terminal and the entities that compose it.

Another relevant framework is JQUERY ([7]) also implemented as an Eclipse plug-in. Similar to JTRANSFORMER, JQUERY is a model generator for JAVA. However, there are major differences between JQUERY and JTRANSFORMER that favor the latter. JQUERY does not model the entire source code of the target system, limiting user queries to interfaces, calls and field accesses while JTRANSFORMER models the entire system. Furthermore, JQUERY is hardly scalable to large software systems (more than 500 classes) and only supports user queries. On the other hand, JTRANSFORMER is highly scalable being tested on software systems with more than 11000 classes and offers both support for analyses and transformations on the model by means of the CT mechanism (conditional transformations).

[12] presents SOUL (Smalltalk Open Unification Language), a language integrated into Smalltalk environments

and dedicated to declarative meta-programming. Using this language one can reason about programs written in Smalltalk, Java or C by using special libraries of logic predicates implemented in SOUL: LiCoR (a library to process Smalltalk programs), Irish (a library to process Java programs) and Zombie (a library to process C programs). [14] describes and application where SOUL is used as a modeling language to detect a number of bad smells ([5])

Finally, [3], [9] and [8] all report code analyses performed on logic meta-models that abstract C++ sources ([3]) or Java sources ([9], [8]). All these meta-models are not exhaustive, they do not model the entire software system under analysis, but only those aspects required by the desired analyses.

6 Conclusions and perspectives

In this paper we have presented PROGEN, a language independent software tool capable of producing PROLOG representations for programs written in any programming language. Such PROLOG representations could be used further to perform program analyses or even program transformations, if another tool is provided that is capable of regenerating the initial sources back from the PROLOG model.

Since our approach is syntax-driven, it can assume almost nothing about the relations between the entities it encounters. The only thing it can assume is the direct relationship between the Prolog fact that describes the left part of an EBNF production and the Prolog facts that describe its right part. However, a model user that is aware of the underlying programming language could easily use this direct relationship and write his own rules to create complex relationships between program entities as required by the concrete programming language for which a model has been created. In other words, the PROLOG model could be adapted or even extended to suit user-specific needs. The main drawback here is that this adaptation requires a good knowledge of the PROLOG language but we believe this is a small price to pay for the advantages it provides.

As future work, we plan to study the possibility of eliminating or reducing the clause chains from the generated PROLOG models (see section 4.3). Another natural research direction would be aimed at regenerating the original source code from the model by consulting the PROLOG facts while also following the language grammar. Thus, the source code regeneration would be a grammar-driven and a fact-driven process.

References

[1] The ISO/IEC 14977:1996(e) standard.

[2] C.-B. Chirilă, C. Jebelean, and A. Măduţa. Towards automatic generation and regeneration of logic representation for object-oriented programming languages. In *Proceedings of the International Conference on Technical Informatics - To be published*, University Politehnica Timişoara, 2008.

[3] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *Proceedings of the Technology of Object-Oriented Languages and Systems*. IEEE Computer Society, 1999.

[4] E. Foundation. Eclipse. http://www.eclipse.org, 2008.

[5] M. Fowler. *Refactoring: Improving the design of existing code*. Addison-Wesley, 2000.

[6] T. R. Günter Kniesel, Jan Hannemann. A comparison of logic-based infrastructures for concern detection and extraction. In *Workshop on Linking Aspect Technology and Evolution (LATE'07), in conjunction with Sixth International Conference on Aspect-Oriented Software Development (AOSD.07), March 12-16, 2007, Vancouver, British Columbia*. Workshop on Linking Aspect Technology and Evolution (LATE'07), in conjunction with Sixth International Conference on Aspect-Oriented Software Development (AOSD.07), March 12-16, 2007, Vancouver, British Columbia, Mar 2007.

[7] D. Janzen and K. de Volder. Navigating and querying code without getting lost. In *Proceedings of the 2nd International Conference on Aspect Oriented Software Development (AOSD'03)*, pages 178–187. ACM Press, 2003.

[8] C. Jebelean. Automatic detection of missing abstract-factory design pattern in object-oriented code. In *Proceedings of the International Conference on Technical Informatics*, University Politehnica Timişoara, 2004.

[9] S.-U. Jeon, J.-S. Lee, and D.-H. Bae. An automated refactoring approach to design pattern-based program transformations in Java programs. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference*, 2002.

[10] G. Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, Jan 2006.

[11] R. Marinescu. Detection strategies: metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, pages 350–359. IEEE Computer Society, 2004.

[12] K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *Special Issue of Elsevier Journal on Expert Systems with Applications*, volume 23, pages 405–413, 2002.

[13] S. Microsystems. JavaCC. https://javacc.dev.java.net, October 2006.

[14] T. Tourwé and T. Mens. Identifying refactoring opportunities using logic meta programming. In *Proceedings of the 7th European Conference on Software Maintainance and Reengineering*, pages 91–100. IEEE Computer Society, 2003.