CONTI'2008
The 8th INTERNATIONAL CONFERENCE ON TECHNICAL INFORMATICS, 5-6 June 2008, TIMISOARA, ROMANIA

Towards Automatic Generation and Regeneration of Logic Representation for Object-Oriented Programming Languages

Ciprian-Bogdan Chirila* and Călin Jebelean** and Anca Măduţa***

*Department of Computer Science, University Politehnica Timişoara, E-mail: chirila@cs.upt.ro, WWW: www.cs.upt.ro/~chirila
**Department of Computer Science, University Politehnica Timişoara, E-mail: calin@cs.upt.ro WWW: www.cs.upt.ro/~calin
***Department of Computer Science, University Politehnica Timişoara, E-mail: ancamaduta@waterford.org

Abstract* – *Logic based representation has great potential for program analysis and transformation. Such a logic support for a programming language can be manually provided by modeling the grammar and writing a parser using semantic actions. Automatic augmentation of a target grammar with specific semantic actions will determine the generation of logic facts from the program AST (Abstract Syntax Tree) and also provide language independency as long as the grammar is kept generic. Such an approach would be useful for any programming language specified by a grammar. This paper presents an approach towards reaching this goal and also discusses potential problems.

Keywords*: *logic based representation, Prolog factbase, program transformation

```
class Person
{
  private int age;

  public int getAge()
  {
    return age;
  }
}
```

Fig. 1. Code Example

I. INTRODUCTION

Logic based models may be used for representing, analyzing and transforming programs and models [10]. Logic models favor program analysis, design flaws detection [2], [9], [12], [8], design pattern mining [7], program transformations [10], [1]. One of the most commonly used language for logic based representations is Prolog.

The logic representation of programs and models has several advantages because Prolog is a well-known programming language and the model manipulation is more natural using declarative Prolog clauses. For example the refactoring process can be manual or locally assisted in development tools like Eclipse [3] because it relies on AST internal representation. Using logic representation we can perform refactorings globally in the whole project. Thus translating a program into logic based representation offers a higher degree of transformation capability.

Transformation tools like [10], [1] dealing with transformations on Java and Eiffel code integrate manually written parsers that generate the logic representation as Prolog facts and unparsers for getting the source code from the model. In order to automatically generate such parsing tools and to extend the number of languages represented using logic models we propose rules to augment the grammars of those languages with specific actions. The goal of this paper is to show that such automations are possible in principle and to put the bases for an implementation.

The paper is organised as follows. In the second section we present some details about the logic based representation. In the third section we present how programs are translated into logic based representations. The fourth section discusses further aspects about the generation process and identifies potential problems. Section five presents several implementation related issues. Section six presents related works. Finally, in section seven conclusions are drawn and future research directions are discovered.

II. LOGIC BASED REPRESENTATION

Program modeling using logic has proved a viable alternative to relational databases with respect to specification and execution of complex analyses and transformations on the source code. Logic-based implementations manifest a certain expresiveness in holding and retrieving information, not easily found in imperative languages. Another advantage lies in the fact that the modeling language (Prolog) is also used to specify analyses and/or transformations on the model (factbase), so once the model is obtained, the whole activity can be moved at the logic layer.

Logic-based approaches are presented in [2], [12] and [9] and will be mentioned in section VII. However, it is the work of [10] that was of particular interest and consisted a starting point for our research. [10] defines a logic-based representation of models and programs as a generalized abstract syntax tree, which is basically a typed labelled graph. Each node has a unique identifier, a type, a set of attributes and a set of edges pointing to other nodes and all these pieces of information can be combined as a clause in a logic language such as Prolog. The unique identifier associated with each clause is automatically generated by incrementation. All these clauses can then be easily manipulated using the same logic language in order to specify analyses on the model or to perform model transformations.

Let us consider the example presented in figure 1. A simplified Prolog representation will present facts for each node in the AST of the program. In this example, we are only

```
class(1,0,'Person',[2,3]).
field(2,1,int,'age',null).
method(3,1,'getAge',[],int,[],4).
block(4,3, ... ).
```

Fig. 2. Factbase Example

interested in facts regarding major entities of the program, like classes, fields and methods, but a logic representation would normally model every single entity.

By inspecting the knowledge base, one can find the name of the class and details about its intimity. For example (see figure 2), the first fact links the class named *'Person'* with the list *[2,3]*. This means the class has two members and their respective IDs are 2 and 3. Inspecting the knowledge base further, one can detect there is a field with ID=2 named *'age'* and a method with ID=3 named *'getAge'*. The corresponding facts present additional information for them which is useful to navigate further in the hierarchy. It is important to notice that each fact in the knowledge base has its unique ID as first argument and the ID of its parent as second argument.

However, it is possible to model such class-field and class-method relations only if we know for a fact that the target language is Java. A software engineer determined to analyze or transform Java code would need a Prolog representation that easily links classes to their fields and methods because these relations are specific to Java and are language dependent. If we deal with a programming language in general and we only have its grammar specification, a Prolog representation would only relate child nodes to their parent nodes in the AST. This is possible by assigning IDs to each node and generating Prolog facts using a grammar-driven approach that relate the ID of a child node in the AST to the ID of its parent. We lose expressivity but since this approach is based on the grammar only it becomes language independent.

Once the Prolog model is generated for a program, a software engineer that is aware of the programming language could easily write his own rules to ease his access to program entities. An automatic approach to this problem is impractical, if not impossible.

III. FACTBASE GENERATION BY GRAMMAR AUGMENTATION

In this section we present a solution for automating the generation of logic representation for a given program starting from its grammar rules. In order to build such a generator, we augment the grammar with generative semantic actions and feed it to a parser generator like Flex [4], Bison [5] or JavaCC [11]. However the main problem is to automatically obtain such an augmented grammar.

We intend to show how an augmented grammar can be built for a programming language by adding semantic actions to its original grammar rules. While parsing a source file written in that language, semantic actions generating logic representations will be executed. Parser generators like Flex, Bison, JavaCC allow attaching semantic actions to each rule in the grammar and those actions may be used for our model generation purposes.

```
Prog ::= "program" <IDENT> ";" Block "."
Block ::= Decls "begin" InstrList "end"
Decls ::= "var" VariableList ":" Type ";"
VarList ::= Var "," VarList | Var
Var ::= <IDENT>
Type ::= "integer" | "real" | "char"
InstrList ::= Instr ( ";" Instr )*
Instr ::= Assignment
Assignment ::= <IDENT> ":=" Expression
Expression ::= <IDENT> | <INT>
```

Fig. 3. MiniPascal Grammar

However, we model the program without losing any information in order to be able to regenerate the original source code. Such a facility will allow program transformations by modeling the program, transforming the model and regenerating the source code back from the model. Model transformation is far more easier than source code modification since Prolog has built-in predicates for factbase manipulation, like adding, removing, modifying clauses.

Semantic actions associated with different grammar rules appear to be quite similar. Their main purpose is to generate for each AST node a Prolog fact that links the ID of the AST node with the IDs of its children, as stated by the corresponding grammar rule.

We intend to augment with semantic actions a simple MiniPascal grammar.

In figure 3 we used both BNF productions and regular expressions to define grammar rules, since many parser generators accept them both. In order to generate semantic actions we issue the following guidelines:

1) Each grammar rule must be augmented with fact-generating semantic actions.
2) The fact corresponding to a rule has its own ID as first argument and also the ID of the parent as second argument. Such a convention will favor factbase navigation.
3) For a grammar rule having alternatives we will generate separate semantic actions. Such a rule is equivalent with multiple rules without alternatives.
4) For each atom we will generate a fact containing a parent reference and its value.

We show examples of how the grammar will be augmented for only four of the rules: *Block*, *VarList*, *Type* and *InstrList*. The *Block* rule is an ordinary rule, the *VarList* rule is a recursive one, the *Type* rule has alternatives and the *InstrList* makes use of a regular expression to simulate recursivity.

In figure 4 we modeled the *Block* instruction from the grammar in figure 3. The rule is composed out of two subrules and two atoms. The semantic actions will create the *block* fact and two facts for the *begin* and *end* atoms, while the

```
Block(int parentId) ::=
    {
      int newId=generateUniqueId();
      output( block(newId,parentId). )
    }
  Decls(newId)
  "begin"
    {
      output( atom(newId,'begin'). )
    }
  InstrList(newId)
  "end"
    {
      output( atom(newId,'end'). )
    }
```

Fig. 4. Block Rule Augmentation

```
VarList(int parentId) ::=
    {
      int newId=generateUniqueId();
      output( varList(newId,parentId). )
    }
Var(newId)
  ","
    {
      output( atom(newId,','). )
    }
  VarList(newId)

VarList(int parentId) ::=
    {
      int newId=generateUniqueId();
      output( varList(newId,parentId). )
    }
Var(newId)
```

Fig. 5. VarList Rule Augmentation

facts corresponding to non-terminals will be handled at their corresponding level.

In figure 5 one of the two rules is right-recurrent. The recursive call will determine *varList* facts linked together in ancestor-descendant relation.

In figure 6 there are three symmetrical alternatives for the *Type* rule. Each one generates a fact for the rule and one for the atom.

In contrast with the right-recursive rule depincted in figure 5 generating facts linked one to another in a row, the *InstrList* rule from figure 7 uses regular expression to simulate the same recursive behavior generating facts linked to a single parent fact *instrList*.

```
Type(int parentId) ::=
    {
      int newId=generateUniqueId();
      output( type(newId,parentId). )
    }
  "integer"
    {
      output( atom(newId,'integer'). )
    }
Type(int parentId) ::=
    {
      int newId=generateUniqueId();
      output( type(newId,parentId). )
    }
  "real"
    {
      output( atom(newId,'real'). )
    }
Type(int parentId) ::=
    {
      int newId=generateUniqueId();
      output( type(newId,parentId). )
    }
  "char"
    {
      output( atom(newId,'char'). )
    }
```

Fig. 6. Type Rule Augmentation

```
InstrList(int parentId) ::=
    {
      int newId=generateUniqueId();
      output( instrList(newId,parentId). )
    }
  Instr(newId)
  (
    ";"
      {
        output( atom(newId,';'). )
      }
  Instr(newId)
  )*
```

Fig. 7. InstrList Rule Augmentation

```
01 program test;
02 var a:integer;
03 begin
04  a:=5
05 end.
```

Fig. 8. MiniPascal Example

```
01 program(10000).
02 atom(10000,'program').
03 atom(10000,'test').
04 atom(10000,';').
05 block(10001,10000).
06 decls(10002,10001).
07 atom(10002,'var').
08 varlist(10003,10002).
09 var(10004,10003).
10 atom(10004,'a').
11 atom(10002,':').
12 type(10005,10002).
13 atom(10005,'integer').
14 atom(10002,';').
15 atom(10001,'begin').
16 instrlist(10006,10001).
17 instr(10007,10006).
18 assignment(10008,10007).
19 atom(10008,'a').
20 atom(10008,':=').
21 expression(10009,10008).
22 atom(10009,'5').
23 atom(10001,'end').
24 atom(10000,'.').
```

Fig. 9. Generated Factbase

IV. SIMPLE USE CASE

In figure 8 we proposed a MiniPascal code snippet which conforms to the grammar listed in figure 3. The example contains a block declaration which is composed out of a variable declaration and an enclosed instruction list.

The factbase generator parsing the example from figure 8 should produce the factbase listed in figure 9. At runtime each rule from the grammar will produce an AST node and for each node a Prolog fact will be generated.

The Prolog facts from figure 9 represent the equivalent model of the MiniPascal example. The first fact from line *01* has no parent and models the program rule from the grammar. The associated atoms of the first grammar rule are listed between lines *02-04* and they point to their parent rule through identifier *10000*. Next, all Prolog facts modeling grammar rules have as first argument their own global identifier and as second argument the identifier of their parent (figure 9 lines *05, 06, 08, 09, 12, 16, 17, 18, 21*). The assignment statement from figure 8 line *04* if translated in the factbase by the assignment fact from line *18* pointed by atoms from lines *18* and *19*. The constant value *5* of the assignment is modeled as an expression and an atom (lines *21* and *22*).

```
Conditional ::=
  "if" Expression "then" InstrList
                  "else" InstrList
```

Fig. 10. Multiple Identical Children Rule

```
conditional(id,parentId,
  instrListId1,instrListId2).
```

Fig. 11. Multiple Identical Children Fact

V. FURTHER ASPECTS

A. Atom Modelisation

A slight difference could be made between terminal nodes regarding the cardinality of their class. We consider that terminals belonging to a class with cardinality = 1 could be ignored since they don't belong to the AST. For example, the terminal ";" that belongs to the class SEMICOLON could not be modeled, because its class cardinality is 1. On the other hand, the atom "name" should be modeled since it belongs to the class IDENTIFIER whose cardinality is greater than 1. However such an approach meant to simplify the model would complicate the unparsing process instead. Rules having alternatives that only differ by one or more atoms would make the unparsing impossible because of the atom(s) information loss.

B. Multiple Children of the Same Class

When modeling rules which contain subrules of the same kind the regeneration is a problem since their order can not be determined from the factbase. A solution for this problem would be the bidirectional navigation so the parent fact will keep references as arguments to its children. A representative example is a *conditional* instruction rule which contains two instruction lists: one for the *then* branch and another one for the *else* branch, like in figure 10.

The fact for this rule must contain two extra arguments the IDs pointing to the two *InstrList* facts like in figure 11. We can generalize this practice for multiple identical subrules. Each time the relative order of the identical subrules will correspond to the argument order in the rule. In the case of multiple groups of identical subrules the order of arguments could become quite complex.

Such problematic rules could be automatically detected by checking the duplicated subrules. The child references for the parent could be added only after parsing the descendants and getting their IDs.

```
instrList(id,parentId,
  [instrId1,...,instrIdn]).
```

Fig. 12. Fact Ordering Example

C. Fact Ordering

A similar problem occurs for mutiple facts generated by the same rule and having a common parent. For example the order of the instructions in an instruction list is crucial for the correct regeneration of the original code. The solution is to keep a list of orderred references to the children at the parent level.

In figure 7 the order of the generated *instr* facts is necessary for the correct regeneration. A solution in this sense is to group the IDs of the child facts into an orderred list like in figure 12.

In figure 12 the generated fact contains an extra list of orderred ID references which will capture the instruction order from the original source code.

The detection of such rules can not be made automatically. One simple solution would be to manually mark the rules which require bidirectional navigation. A general rule that will create bidirectional navigation for any recursive rule may be issued also as partial solution for the given problem.

D. Left/Right Recursion

Regular programming language grammars use left or right recursion in order to model lists of entities. The augmentation of such rules will cause no problems as is was shown in figure 5. Depending on how the rules are expressed the facts will be linked to a single parent as in figure 7 or to one another like in 5. For switching from a representation to another Prolog rules must be manually written.

VI. SOFTWARE INSTRUMENTATION

In order to prove the feasibility of the approach, we set the bases for an implementation. There are several parser generators which accept augmented grammars. Flex [4] and Bison [5] have several disadvantages like: they are not both object-oriented, they use separate grammars for the lexical and syntactical rules, they do not have an online grammar repository. On the other hand Bisson is a LALR parser generator and covers a majority of programming languages. Another solution is to use the JavaCC [11] parser generator which generates recursive descendant parsers, is fully object-oriented (AST nodes and visitors) and has a great repository of programming language grammars including its own grammar format.

VII. RELATED WORKS

The main work we should mention here is the JTransformer framework presented in [7] since our approach is inspired from it. JTransformer produces a very accurate Prolog model of a software system written in Java. Being dedicated to Java, the output model is aware of certain program relations that are specific to this programming language. For example, each Prolog representation of a class definition also contains a list of references to program entities that are members in that class (attributes and methods) and each representation of a method definition contains a list of references to the exceptions that the method raises. Such relations can only be established by a language dependent parser with the sole purpose of facilitating programmer's access to the complete data of a program item. In contrast, our approach is language independent, being able to produce Prolog models for programs written in any programming language. Since it is grammar-guided, 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 a BNF 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.

Logic metaprogramming is widely used to specify code analyses and program transformations. For instance, [2] presents an approach where logic metaprogramming is used to abstract C++ sources and detect certain design flaws. An example is given where a simple Prolog rule is able to detect classes that "know" about their subclasses, which is regarded as a serious design problem. In [12] a variant of Prolog called SOUL is used as a modeling language to detect a number of bad smells. For example, a rule is proposed that detects methods with unused parameters. These methods can be reported as potential candidates for refactorings that remove the unused parameters. [9] and [8] both use Prolog rules to detect suspect object instantiations within Java source code and recommend an Abstract Factory ([6]) solution where needed. The approaches are quite similar, however [8] presents a more accurate detection strategy by considering the control paths of methods.

Regarding program transformations, in [1] the transformations are used to implement the semantics of reverse inheritance - a class reuse mechanism based on an inheritance class relationship where subclasses exist first and the superclass is created afterwards.

All these works could make use of the methodology we propose in this paper. However, as mentioned earlier, language independency specific to our approach will definitely complicate the implementation of program specific queries. Our grammar-driven logic based model should be enhanced with language specific rules that would eventually ease the specification of analyses and code transformations.

VIII. CONCLUSIONS AND PERSPECTIVES

In this work we proposed an approach for a grammar driven generation of facts from an AST. The approach was designed to work automatically on multiple programing languages (Java, Eiffel, C++) whose rules are expressed in a grammar.

This approach can be applied also to models which are described by grammar rules not just to programs. Also, we consider that grammar of common PLs is always available. In order to perform program transformations on the generated model, Prolog skills are required.

As drawback of the approach, the grammar could be quite complex, resulting a complex metamodel to be handled. In some programming languages grammars, some rules are artificially added in order to be compliant with the parser generator. Because the generation process is grammar-driven, facts for such rules will be automatically generated. In order to bypass such levels, special access Prolog rules must be written manually.

By default the generated facts in logic representation are designed to point to their parents, thus alowing upward navigation. Downward navigation may be necessary sometimes for the relation between facts, but such things may be automatically generated if marked or otherwise such navigation constructs have to be written manually using Prolog rules. The resulting model with its conventions will offer an ideal support for writing language independent rules.

As main future work comes naturally the regeneration of the original source code from the model. Such a regeneration implies consulting the Prolog facts upon the language grammar. The regeneration is a grammar driven and fact driven process.

In order for a complete regeneration to be possible, the modeling process is responsible for storing all the information from the original program sources.

However, some details regarding spacing and identation will not be possible to perform. For such purposes ah-hoc rules may be set or solved by manually written rules completing the syntax.

Following the same ideas a type checking rule list could be generated in order to check model validity. Such a rule list will consist in describing each fact argument type to restrict possible relations between them. An implementation for an automatic checker is under development [10].

REFERENCES

[1] Ciprian-Bogdan Chirila, Gunter Kniesel, Philippe Lahire, and Markku Sakkinen. Eiffel/RI Project Website. http://nyx.unice.fr:9000/trac, December 2007.

[2] Oliver 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.

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

[4] Free Software Foundation. Flex - a fast scanner generator. http://www.gnu.org/software/flex, March 1995.

[5] Free Software Foundation. Bison - GNU parser generator. http://www.gnu.org/software/bison, 2006.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley, 1995.

[7] Tobias Rho Gnter 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.

[8] Călin 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] Sang-Uk Jeon, Joon-Sang Lee, and Doo-Hwan 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] Günter Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, Jan 2006.

[11] Sun Microsystems. Java compiler compiler. https://javacc.dev.java.net, October 2006.

[12] Tom Tourwé and Tom 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.