Language-Independent Generation of Logic Representations for Programs

CĂLIN JEBELEAN
University ”Politehnica” Timişoara
V. Pârvan no. 2, Timişoara
ROMANIA
calin.jebelean@cs.upt.ro

CIPRIAN-BOGDAN CHIRILA
University ”Politehnica” Timişoara
V. Pârvan no. 2, Timişoara
ROMANIA
ciprian.chirila@cs.upt.ro

TITUS SLAVICI
University ”Politehnica” Timişoara
V. Pârvan no. 2, Timişoara
ROMANIA
titus.slavici@mec.upt.ro

VLADIMIR CREŢU
University ”Politehnica” Timişoara
V. Pârvan no. 2, Timişoara
ROMANIA
vladimir.cretu@cs.upt.ro

Abstract: Logic representation is a strong foundation for program analysis and transformation. Obtaining metamodel conforming logic representation for programs written in general purpose programming languages requires a generic methodology. The declaration of a logic metamodel using generic AST node data access expressions facilitates automatic generation of logic representation without requiring knowledge of advanced program transformation tools.

Key–Words: program transformation, semantical actions , metamodel conforming logic representation

1 Introduction

A logic representation for a program is a suitable way of dealing with the inherent complexity of certain problems like program analysis and program transformation. It is generally accepted that declarative languages are more expressive than imperative languages in this regard. Figure 1 shows this approach:

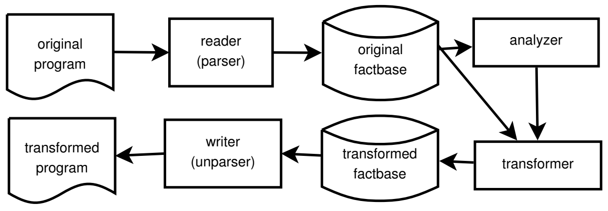

Figure 1: Logic Based Approach

In [6] the authors present a tool called JTransformer which is capable of transforming Java programs into Prolog facts. All Java entities are modeled in Prolog and navigation means are also provided. Thus, one can access in Prolog all the packages of the Java project, for each package a list of classes can easily be obtained, each class provides access to information about its attributes and methods, and so on. By using the inference power of Prolog it is easy to analyze the original Java code or even modify the logic facts such as to add or delete a class, add or delete a method, or even correct some of the problems found by the analysis step that we have mentioned before. Currently, programming language logic representation was succesfully used for design antipattern detection ([4]) and in the implementation of a reverse inheritance class reuse mechanism for Eiffel ([10]).

[5] is inspired from the previously mentioned work and shows how simple representations in Prolog can be obtained for programs written in any language. Using the grammar for the target language, a collection of Prolog facts is generated by following the structure of the abstract syntax tree (AST) of the program. Each father-son relation in the AST is modeled as a fact in Prolog. The representation is not quite useful for analysis and transformation because it is much too close to the grammar of the target language and does not provide easy access to relations between program entities. For example, obtaining attribute or method information for a given class (if the target language is an object-oriented one) is not an easy task. Since the approach in [5] is a language independent one (this means the target language can be any language), such a desiderate would be quite ambitious, because the grammar only provides syntactic information about the language. Using only the language grammar it is hardly possible even to detect if the respective language is object-oriented.

In this paper we are going to combine the two approaches presented before. Namely, we will show how JTransformer-like output can be obtained for programs written in any language.

The paper is structured as follows: section 2 presents the logic based representation of programs, section 3 describes the language independent program translator, section 4 walks through the proposed methodology using examples, section 5 studies related works and section 6 concludes and sets perspectives.

2 Logic-Based Representation of Programs

Logic based representation of programs is all about writing Prolog facts that encapsulate in one form or another all the information available in the original program. Logic facts are linked to one another by means of their unique integer identifiers, creating a hierarchical structure much like a generalized tree. Later we will show that this generalized tree is, in fact, similar to the abstract syntax tree of the program. There should be one root fact which points to its children by specifying a list of their identifiers, while its children all point to the root fact also by linking to its identifier. This idea is preserved at all levels. A logic fact will then look like this (first three parameters of each fact are only for identification and navigation):

```
factName(<id>, <pid>, <cids>, <argument> ...).
```

where:

- <id> is the integer identifier of the current fact
- <pid> is the integer identifier of the current fact's parent
- <cids> is a list of integer identifiers of the current fact's children
- <argument> is the first argument of the current fact and can be followed by others

For exemplification we present the Java code in figure 2:

```
class Rectangle {
  private double width;
  private double height;

  public double area() {
  }
}
```

Figure 2: Sample Code

Representing such a program in Prolog using the elements introduced earlier is straightforward. The root fact will define the class, while child facts will deal with attributes and methods. Each of these childs will have childs of its own that further describe child-related aspects, like type information and access modifiers for fields and methods, for example.

```
classDef(100, 0, [101, 104, 107], 'Rectangle').
fieldDef(101, 100, [102, 103], 'width').
accessDef(102, 101, [], 'private').
typeDef(103, 101, [], 'double').
fieldDef(104, 100, [105, 106], 'height').
accessDef(105, 104, [], 'private').
typeDef(106, 104, [], 'double').
methodDef(107, 100, [108, 109], 'area').
accessDef(108, 107, [], 'public').
typeDef(109, 107, [], 'double').
```

Figure 3: Sample Logic Representation

The logic representation is quite expressive and easy to use. The class definition (id 100) offers quick access to class members (ids 101, 104 and 107) which turn out to be two fields and a method. Each of them offers quick access to information such as the access modifier and the type.

3 Language Independent Logic Generation

The current limitation is that Prolog output like the one in figure 3 can only be obtained for specific languages, by manually writing suitable translators aimed at those respective languages. Such a Prolog transformation engine is JTransformer ([6]) which only deals with Java programs. This article introduces and describes a new methodology which we intend to implement in a tool called ProGen aimed at performing language-independent generation of logic facts for programs. In order to be able to generate such logic facts for programs written in any language, our new methodology should be based on two things:

- the grammar of the respective language – a crucial artifact, since it is the only one that can provide information about language elements and how they relate to each other;
- a set of mapping rules attached to each grammar production that specify what needs to be done at each point.

[5] presents a naive version of such a methodology. It is language-independent but it is also naive because it only uses the grammar for generating logic facts, without the previously mentioned mapping rules. Thus, the output only contains instances of father-son relations directly extracted from the grammar. The main problem is that high-level relations

between program elements are very difficult to grasp from the Prolog output unless they are directly contained in the grammar. However, the fact that a class in an object-oriented language contains fields and methods is not something directly specified in the grammar, but rather deductible by following a number of grammar rules. Such high-level relations between program entities must be specified by some kind of annotations which should accompany the language grammar. Basically, the resulting Prolog model must be an instance of a metamodel that should specify in detail how program items are related to each other. This is what the mapping rules introduced earlier actually are. The approach augments the one in figure 1 with the grammar and metamodel mapping rules, as described earlier and is presented in figure 4. The grammar and the metamodel mapping rules are used together with JavaCC ([8]) to generate a special parser for programs written in a language that conforms to the grammar. Replacing the grammar and the metamodel mapping rules will lead to the same methodology being applied to another programming language.

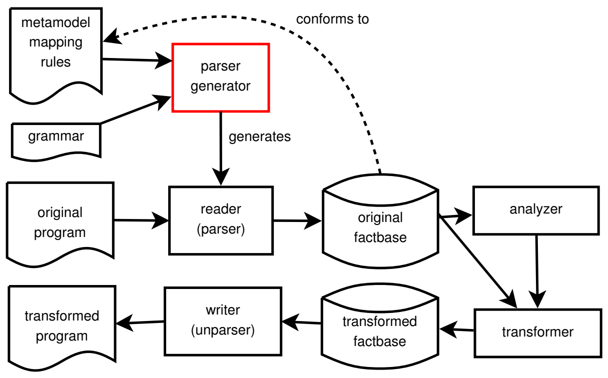

Figure 4: ProGen Approach

The parser in figure 4 can be generated by using the language grammar (which is available, as mentioned) and JavaCC which is a parser generator that generates parsers written in Java. The fact generation behavior can be attached to the parser by using a Visitor design pattern ([2]) which is instructed to generate Prolog facts in every AST node it visits. The instructions for fact generation are the mapping rules in figure 4.

4 A Methodology Walkthrough

In this section we will show how things work by presenting a comprehensive example which will include a simple grammar and a simple set of mapping rules for that grammar. The target language will be Eiffel ([7]) but the approach is language-independent anyway.

4.1 Grammar Rules

Our working grammar is presented in figure 5. It describes a simplified version of the syntax of class and feature declarations in Eiffel (note that in Eiffel, both fields and methods are referred to as features):

```
ClassDecl ::= [ "deferred" ] "class" <id> ( FeatureBlock )* "end"
FeatureBlock ::= "feature" ( FeatureDecl )*
FeatureDecl ::= <id> [ "(" FormalArgumentList ")" ]
                [":" <id>] [ FeatureBody ]
FormalArgumentList ::= FormalArgument ( ";" FormalArgument )*
FormalArgument ::= <id> ":" <id>
FeatureBody ::= ...
```

Figure 5: Eiffel Grammar Excerpt

We will ignore everything under FeatureBody (the body of features) because of space limitations.

4.2 JavaCC Library Extension

As mentioned earlier, the mapping rules which will be the subject of the next subsection will use generic AST node data access expressions. Specifically, they will use concepts like: the identifier of an AST node, the n-th direct descendant of an AST node, the parent of an AST node, the next sibling of an AST node, etc. JavaCC provides access to some of this data, but not for all. Moreover, JavaCC treats lexical tokens separately from AST nodes. Lexical tokens are instances of class Token, while AST nodes are objects of descendant types of class SimpleNode. By default, there is no relation between class Token and SimpleNode in the library, but JavaCC provides means to set ancestor for these classes. We decided that it was easier to unify the two concepts by making them inherit from the same superclass, called Entity. The new library (original library plus extensions) is presented in figure 6:

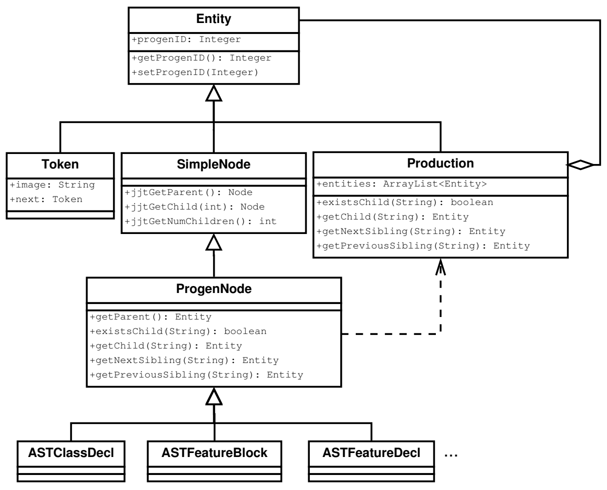
Entity
+progenID: Integer
+getProgenID(): Integer
+setProgenID(Integer)
Token
+image: String
+next: Token
SimpleNode
+jjtGetParent(): Node
+jjtGetChild(int): Node
+jjtGetNumChildren(): int
Production
+entities: ArrayList<Entity>
+existsChild(String): boolean
+getChild(String): Entity
+getNextSibling(String): Entity
+getPreviousSibling(String): Entity
ProgenNode
+getParent(): Entity
+existsChild(String): boolean
+getChild(String): Entity
+getNextSibling(String): Entity
+getPreviousSibling(String): Entity
ASTClassDecl
ASTFeatureBlock
ASTFeatureDecl
...

Figure 6: JavaCC Library Extension

Figure 7 shows the library at work on a simple example:

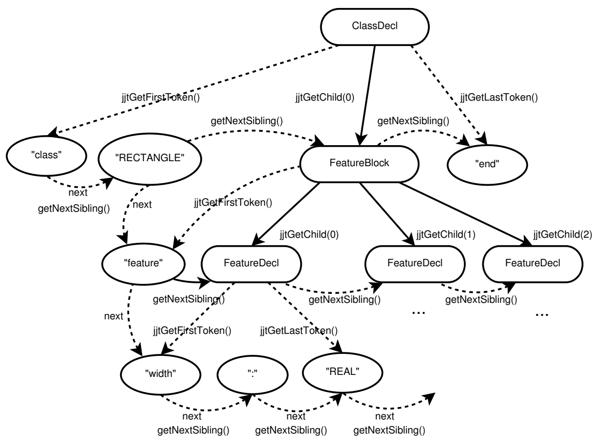

Figure 7: JavaCC Node Navigation Methods

Methods having ”jjt” as prefix are methods already supplied by JavaCC. These methods have some annoying limitations. For example, methods jjtGetChild provides access to a child of a SimpleNode given by its index among the children. However, only children of type SimpleNode are counted and if there are also children of type Token, they are disregarded. We needed a method that would treat Tokens and SimpleNodes in a unified manner, that’s why we introduced a new version of this method called getChild. Also, we needed some methods to obtain the next or the previous sibling of a given child. These methods are called getNextSibling() and getPreviousSibling().

4.3 Mapping Rules

The set of mapping rules defines the structure of the desired metamodel by using expressions built with node operations from the JavaCC extended library described earlier. Mapping rules are divided into two categories: rules that specify what Prolog code does an AST node generate and rules that specify what value does an AST node return to be used by the ancestors of that node.

4.3.1 FormalArgument Mapping Rule

The first example considers the grammar rule which defines a FormalArgument:

```
FormalArgument ::= <id> ":" <id>
```

From the syntactic point of view, a formal argument definition is a sequence of 3 atoms: an identifier, a colon and another identifier. From a semantic point of view, the first identifier is the name of the argument and the second identifier is its type. A metamodel builder who has access to language semantics would know that.

The Prolog facts that would express the same thing would be:

```
formalArgument(#ID, #ParentID, 'name', 'type').
```

To achieve that, we should use the mapping rule in figure 8:

```
FormalArgument generate
  formalArgument(
    node.getProgenID(),
    node.getParent().getProgenID(),
    "'" + node.getPreviousSibling(":") + "'",
    "'" + node.getNextSibling(":") + "'"
  ).
```

Figure 8: FormalArgument Mapping Rule

The mapping rule specifies that a FormalArgument node must generate a formalArgument Prolog fact with 4 parameters: the first one is the ProGen identifier of the node, the second one is the ProGen identifier of the node’s parent, the third one is the previous sibling of the atom ”:” enclosed in simple quotes and the fourth one is the next sibling of the atom ”:” also enclosed in simple quotes.

Now this textual rule must be systematically transformed in a Java visitor method which will be part of the generated parser in figure 4. The systematic transformation is necessary because the mapping rules will be processed automatically and the Java code (visitor and helper methods) will be generated automatically.

```
public Object visit(ASTFormalArgument node, Object data) {
  System.out.println(
    "formalArgument" + "(" +
      node.getProgenID() + "," +
      node.getProgenParent().getProgenID()+ "," +
      "'" + node.getPreviousSibling(":") + "'" +
      "'" + node.getNextSibling(":") + "'" + ")."
  );
  data = node.childrenAccept(this, data);
  return data;
}
```

Figure 9: FormalArgument Visitor Method

A parser that uses a Visitor that calls the method in figure 9 when dealing with an ASTFormalArgument node will generate a Prolog fact according with 4 parameters, as needed. It is easy to observe that the mapping rule in figure 8 can easily be transformed in the Java method from figure 9, all the expressions used to describe parameters should just be ”copied and pasted” in the output Java code in the suitable places. This is one of the most important advantages of the approach we present here. Each mapping rule use a simple syntax that is already compatible with the Java compiler and its integration in a larger Java project (like a parser generator) can be easily automated.

4.3.2 FormalArgumentList Mapping Rule

The FormalArgumentList grammar rule is chosen as the second example because it will give us the chance to explain the second type of mapping rules: rules that specify return values for AST nodes.

```
FormalArgumentList ::= FormalArgument ( ";" FormalArgument )*
```

The Prolog facts that would express the same thing would be:

```
formalArgumentsList(#ID, #ParentID,
  [#FormalArgument1ID, #FormalArgument2ID, ...]).
```

Normally, since a FormalArgumentList contains a number of formal arguments, it could be required to generate a list of IDs of those formal arguments. For that purpose, we will have a mapping rule that defines what value should an ASTFormalArgumentList node return and another mapping rule that defines what Prolog code should an ASTFormalArgumentList generate.

```
FormalArgumentList return list(FormalArgument)

FormalArgumentList generate
  formalArgumentList(
    node.getProgenID(),
    node.getParent().getProgenID(),
    node
  ).
```

Figure 10: FormalArgumentList Mapping Rules

The generated Prolog fact will have 3 arguments: the ProGen identifier of the node, the ProGen identifier of the node's parent and the value of the node itself. This node value is specified by the other mapping rule. The value of an ASTFormalArgumentList node is a list of all ASTFormalArgument nodes that descend from it. These mapping rules will be translated into 2 Java methods. The rule containing the "generate" clause will be translated in the same manner like in the previous example, but for the rule containing the "return" clause we will use a special type of translation. The Java output is visible in figure 11:

Again, the Java code can be obtained mechanically from the mapping rules and can be the subject of automatic generation.

```
public Object visit(ASTFormalArgumentList node, Object data) {
  System.out.println(
    "formalArgumentList" + "(" +
    node.getProgenID() + "," +
    node.getProgenParent().getProgenID() + "," +
    computeReturnValue(node) + ")."
  );
  data = node.childrenAccept(this, data);
  return data;
}

public String computeReturnValue(ASTFormalArgumentList node) {
  String szValue = "[";
  for(int i = 0; i < node.jjtGetNumChildren(); i++) {
    ProgenNode pgNode=(ProgenNode)node.jjtGetChild(i);
    if (pgNode instanceof ASTFormalArgument)
      szValue += pgNode.getProgenID() + ",";
  }
  // cutting the trailing comma
  if(szValue.length() > 1)
    szValue = szValue.substring(0, szValue.length() - 1);
  szValue += "]";
  return szValue;
}
```

Figure 11: FormalArgumentList Visitor Methods

5 Related Works

In this section we will focus on the most representative and similar program transformation works related to our proposed methodology.

Stratego/XT [1] is a framework for implementing software transformation systems. Stratego is a language for software transformation based on the paradigm of rewriting strategies. Basic transformations are defined using conditional term rewrite rules. These are combined into transformations by means of strategies, which control the application of rules. The approach is based also on Syntax Definition Formalisms (SDF) and Annotated Terms (ATerm) which is an abstract data type designed for the exchange of tree-like data structures between distributed applications. Our approach is much simpler, it relies on: metamodel design, basic Java knowledge and a few JavaCC API node data access methods.

EMFText [3] is an Eclipse plug-in that allows the definition of language syntax described by an Ecore model. It is designed for textual representation of Domain Specific Languages (DSL). Our work has the same goal of creating models from text. Our models are represented by Prolog facts, while EMFText models are of Ecore based. The EMFText approach works in both ways: text to model (by a parser) and model to text (by a printer), while our methodology is oriented from text to logic model. Both approaches are based on parser generators ANTLR, respectively JavaCC, which are limited to a subset of context-free grammars.

Kermeta-Sintaks [9] is a tool which defines bridges between concrete (textual files) and abstract syntax (models). The bridge is a Sintaks model used to parse a text in order to get the corresponding model (conforming to a metamodel) and to explore a model for printing in into textual representation. Our mapping rules are similar to the text to model transformation. Both approaches generate metamodel driven models one based on Ecore and the other on Prolog.

Other program transformation tools and languages offer specific methodologies for language engineers like: Fermat - industrial strength program transformation system based on WSL language; Design Maintenance System (DMS) - set of industrial tools for complex source program analysis and transformation; Monticore - framework for the design and processing DLS, etc.

6 Conclusions and Perspectives

In this paper we show how a program can be translated into logic representation conforming to a desired metamodel by using a grammar aware approach. The source code was parsed by a JavaCC grammar generated parser that produces the AST of the code. The logic model is obtained by collecting data from the AST following the metamodel rules. To help accessing data from the AST we configured and extended the Java AST nodes with special methods.

Our approach is simple and pragmatic having the determined goal of generating logic representation for object-oriented programming languages. One of the main advantages of this approach is that mapping rules are written in a language that conforms to the Java syntax and even make use of the JavaCC extended library that we implemented. Thus, there is no need to parse the expressions used to specify mapping rules. The automatic generator can copy and paste them in the output Java code and possible errors will be discovered by the Java compiler itself upon compilation of the generated Java sources.

The approach was experimented with success on an industrial strength language, namely Eiffel. The complexity of the Eiffel grammar prevented us from describing the whole process, we only managed to offer a simple idea about the methodology being used.

As perspectives, we plan to implement the automatic generator of semantic actions in a visitor of the abstract syntax tree. We also plan to provide mapping rules for other general purpose programming languages like C++, Java, C#. Another useful feature would be to be able to automatically regenerate the target program from Prolog facts. That way, the full path presented in figure 4 is complete.

References:

[1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.16: components for transformation systems. In *Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 95–99, New York, NY, USA, 2006. ACM.

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

[3] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, and Christian Wende. Derivation and refinement of textual syntax for models. In *Model Driven Architecture - Foundations and Applications*, pages 114–129. June 2009.

[4] 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.

[5] Călin Jebelean, Ciprian-Bogdan Chirila, and Anca Măduţa. Generating logic based representation for programs. In *In Proceedings of 2008 IEEE 4-th International Conference on Intelligent Computer Communication and Processing*, pages 145–151, Cluj-Napoca, Romania, August 2008.

[6] Günter Kniesel, Jan Hannemann, and Tobias Rho. 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).* Vancouver, British Columbia, March 2007.

[7] Bertrand Meyer. Eiffel: The language. http://www.inf.ethz.ch/˜meyer/, September 2002.

[8] Sun Microsystems. Java Compiler Compiler (JavaCC) - the Java Parser Generator. https://javacc.dev.java.net, March 2010.

[9] Pierre-Alain Muller, Franck Fleurey, Frdric Fondement, Michel Hassenforder, Rmi Schneckenburger, Sbastien Grard, and Jean-Marc Jzquel. Model-driven analysis and synthesis of concrete syntax. In *Proceedings of the MoDELS/UML 2006*, Genova, Italy, October 2006.

[10] Markku Sakkinen, Philippe Lahire, and Ciprian-Bogdan Chirila. Towards fully-fledged reverse inheritance in Eiffel. In *In Proceedings of 11th Symposium on Programming Languages and Software Tools SPLST 09 and 7th Nordic Workshop on Model Driven Software Engineering NW-MODE 09*, pages 132–146, Tampere, Finland, 2009.