Generating Logic Representations for Programs in a Language Independent Fashion

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: In today's software engineering program analysis and program transformation are operations that strongly rely on software models. One important share in this direction is held by logic based models, described in a declarative language such as Prolog. There are some approaches used to represent information about software systems while at the same time preserving the logic relations between entities, but they are normally limited to software systems written in a certain programming language. There are also language independent approaches to logic based representation of programs, but they are usualy based on syntactic information about the modeled program and provide little information about the logic relations between entities at the semantic level. This paper describes a methodology that would unite the two kinds of approaches, being both language independent and expressive at the semantic level at the cost of a more complex generation process.

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:

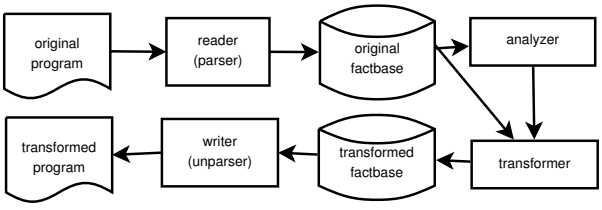

Figure 1: Logic Based Approach

In [12] 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 successfully used for design anti-pattern detection ([9]) and in the implementation of a reverse inheritance class reuse mechanism for Eiffel ([17]).

[10] 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 [10] is a language independent one (this means the target language can be any language), such a desiderata 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 the representation metamodel, section 4 describes the language independent program translator, section 5 walks through the proposed methodology using examples, in section 6 we present two use cases: one dealing with the implementation of a class reuse mechanism and the second with design anti-pattern detection, section 7 studies related works and section 8 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 example 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 children will have children 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 Metamodel

In this section we present the metamodel [11] of the factbase representation for programs. This time will take an Eiffel example. A model which is formally

described by its metamodel offers the reflection facility. Thus, metamodel driven logic models can be easily manipulated at both concrete and meta levels in Prolog. Relying on the metamodel meta-routines can be written to analyze or transform the logic models.

3.1 Node Structure

In figure 4 we present the structure of an AST node and a relation node in the context of logic representation. Specifically, we present the structure of a class declaration and its deferred property from the Eiffel programming language [13, 8]. In Eiffel a deferred class can not have instances, it is the equivalent of a Java abstract class [1].

```
01 %classDecl(#id,#cluster,'ClassName',
02  [#formalGeneric,...]).

03 ast_node_def('Eiffel',classDecl,[
04  ast_arg(id, mult(1,1,no),
05   id, [classDecl]),
06  ast_arg(parent, mult(1,1,no),
07   id, [cluster]),
08  ast_arg(className, mult(1,1,no),
09   attr, [atom]),
10  ast_arg(formalGenerics,mult(0,*,ord),
11   id, [formalGeneric])]).

12 ast_relation('Eiffel',deferred,[
13  ast_arg(classDeclRef, mult(1,1,no),
14   id, [classDecl])]).

15 ast_sub_tree('Eiffel',formalGenerics).

16 ast_ref_tree('Eiffel',classDeclRef).

17 ast_ancestor_tree('Eiffel',parent).
```

Figure 4: Metamodel Example

The class declaration logic representation is defined as a fact with the following arguments:

i) unique global identifier for each AST node;

ii) parent class identifier - refers to the cluster the class belongs to;

iii) class name - as an attribute containing the name of the field;

iv) formal argument list - refers to each child formal generic parameter.

The metamodel AST node is defined by the *ast_node_def* fact which has as first argument the name of the programming language the node refers to (Eiffel), the name of the node (classDecl), followed by a list of arguments describing other details of the AST node. In the context of this work the notion of fact and node are considered to be synonyms. The name of the fact can be considered its type. Each argument is described by a *ast_arg* fact which has properties like:

i) argument name - some names are predefined like *id* or *parent*, but the rest can be freely chosen;

ii) multiplicity - can be zero to one (line 10), one to one (lines 04, 06, 08), zero or many or even one to many;

iii) ordering - it makes sense when multiplicity is zero to many or one to many and in this case the argument is a list which can be ordered or not. For example a class may have zero or more formal generics and their order is important in such cases.

iv) kind of value - it can be identifier (lines 05, 07, 11) or attribute (line 09). Identifiers are positive integers, while attributes are Prolog [5] atoms.

v) legal syntactic type(s) of argument values - there can be one or many AST node types. For example, the type of the identifier argument is *classDecl*, the type of class parent is *cluster*, the type of the formal generics is a type named *formalGeneric*.

The relation node defined between lines *12-14* has only one reference to the parent class. The nature of metamodel AST arguments is defined by special clauses like: i) *ast_sub_tree* for subtree relations (line 15), e.g. *formalGenerics* AST argument denotes the subnodes of *classDecl*;

ii) *ast_ref_tree* for references relations (line 16), e.g. *classDeclRef* AST argument denotes a relation with *classDecl*;

iii) *ast_ancestor_tree* parent relations (line 17), e.g. *parent* AST argument denotes the cluster parent of class declaration.

3.2 Node Arguments

Generic routines which manipulate logic representation depend very much on the fact arguments. We have several kinds of arguments:

i) identifier arguments which represent the unique identifier of the fact (like id for *classDecl*);

ii) parent arguments which represent the parent identifier of the fact (like cluster for classDecl);

iii) ordered list of AST child identifiers (like formal generics for classDecl);

iv) relation arguments which refer other nodes from the structure. Knowing the argument types, generic routines can easily navigate or clone the logic model.

In figure 5 we present an example of an Eiffel class declaration node and its potential relations with other nodes from the model:

i) *classDecl* fact models an Eiffel generic class and has an argument which refers the parent fact *cluster* (Eiffel package of classes);

ii) *classDecl* fact has a list of identifiers pointing towards the two formal generics of the class, identified by *151* and *152*;

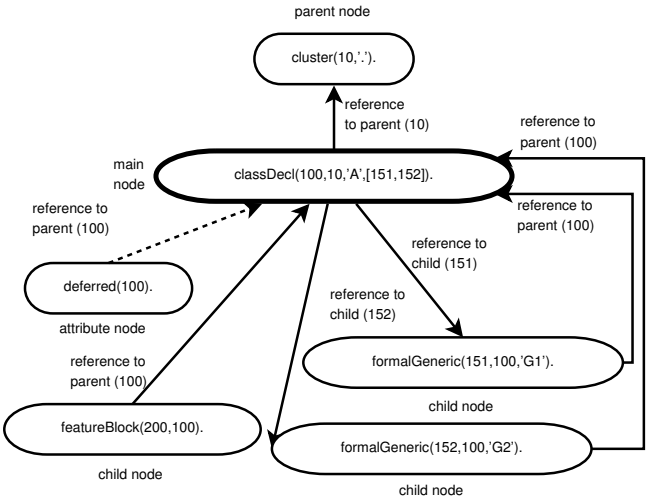

Figure 5: AST Node Structure

iii) both *formalGeneric* facts (*151* and *152*) refer their *classDecl* parent identified by *100*;

iv) *featureBlock* fact is a child of classDecl, but the parent is unaware of its existence;

v) *deferred* fact is a special kind of node, having the role of attribute for *classDecl*, this fact has no own identifier being strongly linked to its parent.

We can notice that the metamodel includes both **syntactical** and **semantical** information. For example the fact that a formal generic belong to a class is a syntactical information. But if we consider a call instruction in the form of *name* or *object.name* it will refer either a formal argument, local or class member (method or attribute), which is a semantical information.

4 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 ([12]) 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.

[10] 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 6. The grammar and the metamodel mapping rules are used together with JavaCC ([14]) 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.

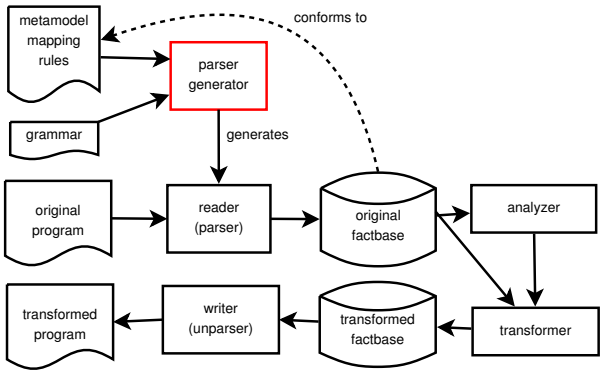

Figure 6: ProGen Approach

The parser in figure 6 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 ([6]) which is instructed to generate Prolog facts in every AST node it visits. The instructions for fact generation are the mapping rules in figure 6.

5 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 ([13]) but the approach is language-independent anyway.

5.1 Grammar Rules

Our working grammar is presented in figure 7. 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 7: Eiffel Grammar Fragment

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

5.2 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 8:

Figure 9 shows the library at work on a simple example:

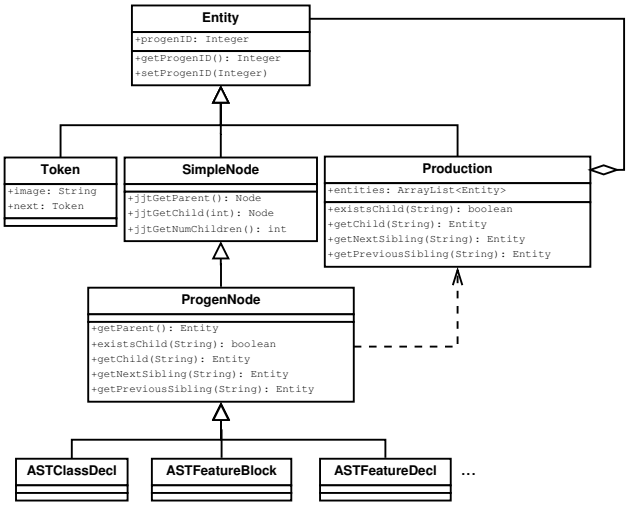

Figure 8: JavaCC Library Extension

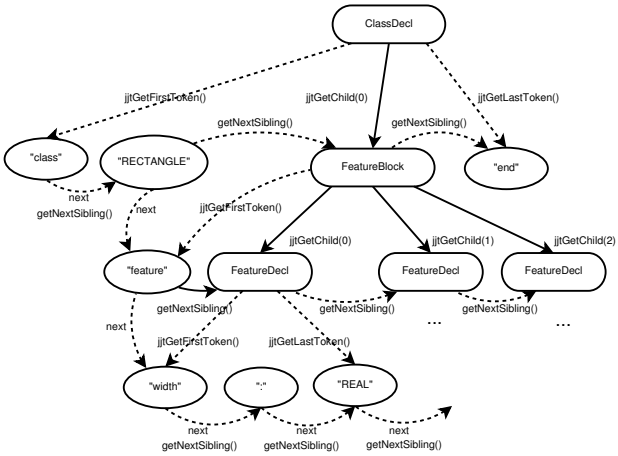

Figure 9: JavaCC Node Navigation Methods

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().

5.3 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 de-

scribed 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. The metamodel presented in section 3 describes the structure of the logic facts. To be noted that the mapping rules must be written according to the metamodel in order to generate the desired logic model.

5.3.1 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 10:

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

Figure 10: 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 6. 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 11: FormalArgument Visitor Method

A parser that uses a Visitor that calls the method in figure 11 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 10 can easily be transformed in the Java method from figure 11, 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.

5.3.2 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.

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

```
FormalArgumentList return list(FormalArgument)

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

Figure 12: FormalArgumentList Mapping Rules

"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 13:

```
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 13: FormalArgumentList Visitor Methods

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

6 Use Cases

6.1 Method Body Exheritance

Informally, reverse inheritance (exheritance) [17] is an inheritance class relationship where the subclasses exist first and the superclass is built afterwards. Reverse inheritance implements the class generalization concept of UML [16]. On the other hand, reverse inheritance is a class reuse mechanism equipped with capabilities like:

- allowing a more natural class hierarchy design - it is more natural to identify the classes in a system, to notify commonalities, to extract them in superclasses;
- common feature factorization from existing classes;
- reusing behavior from a subclass into the superclass and its descendants;
- extending a class hierarchy by creating a superclass and descendants;
- decomposing and recomposing classes with the help of multiple inheritance;
- adding a new layer of abstraction in an existing hierarchy;
- favoring the use of design patters [6].

Eiffel [8, 13] has several language features like: multiple inheritance, no overloading, adaptations, covariance, so it was decided that Eiffel is the most suitable language for the implementation of reverse inheritance. In Eiffel class members both attributes and methods are named as features.

As mentioned earlier, one of the goals for this class relationship is to factor common features from existing subclasses and to create a new representative feature in the foster class. Implicitly, several candidate features from subclasses are exherited as deferred (abstract) in the superclass. The other choice is to explicitly select an implementation from one candidate class and to adjust it to the context of the superclass.

In figure 14 we present two existing classes *RECTANGLE* and *ELLIPSE* and a new class *SHAPE* created by reverse inheritance on the top of them. The first two classes have two common features *perimeter* and *semiperimeter* which are intended to be exherited into *SHAPE* superclass. The decision taken is to exherit *perimeter* as an abstract feature and *semiperimeter* together with its implementation from *RECTANGLE* in order to be reused in other subclasses of *SHAPE*. In order to migrate the implementation from *RECTANGLE* into *SHAPE* we have to take three actions: i) to analyze the *semiperimeter* code from *RECTANGLE* and to search for all calls pointing to class features and to detect if all those calls point to features which were exherited (abstract or concrete); ii) if the condition in i) holds then we clone the feature implementation nodes (body of the method); iii) to replace all local references from *RECTANGLE* with references from *SHAPE*.

In figure 15 we present the result of the local calls search routine. Using generic search rules [4] we extract all calls from the method body subtree. In our case we found only one call to feature *perimeter*. Since the called feature exists in both *RECTANGLE* and *ELLIPSE* subclasses having the same signature, this feature is exheritable in class *SHAPE*. Now we know that the implementation of *semiperimeter* is exheritable and we can proceed to the cloning step.

```
01 class RECTANGLE
02  feature
03   ...
04   perimeter:REAL
05   semiperimeter is
06   do
07    Result:= perimeter/2
08   end
09 end
10
11 class ELLIPSE
12  feature
13   ...
14   perimeter:REAL
15   semiperimeter is
16   do
17    -- ellipse implementation
18   end
19 end
20
21 foster class SHAPE
22  exherit
23   RECTANGLE
24    moveup semiperimeter
25   end
26   ELLIPSE
27 end
```

Figure 14: Method Body Exheritance Example (Eiffel Code)

In figure 16 we duplicated the nodes of the method body in class *SHAPE* using generic cloning rules [4]. One can notice that there is an invalid call reference pointing towards *perimeter* attribute of class *RECTANGLE* instead of the attribute from class *SHAPE*.

In figure 17 we correct all invalid references. Using generic replacement rules [4] and the feature correspondence map from the exheritance process we replace all the invalid ex-local references with the correct ones.

6.2 Antipattern Detection

Another useful purpose for a language-independent logic translator would be to help in the process of antipattern detection using a logic based approach. Design patterns ([6]) are high quality and well documented solutions to frequently occuring problems in object-oriented programming practice. Their use is strongly recommended by experts to help create more robust designs for software applications. However, unexperienced programmers often misuse or even ignore them. A good research direction would be aimed at detecting places in object-oriented code where such design patterns could have been used, but weren't.

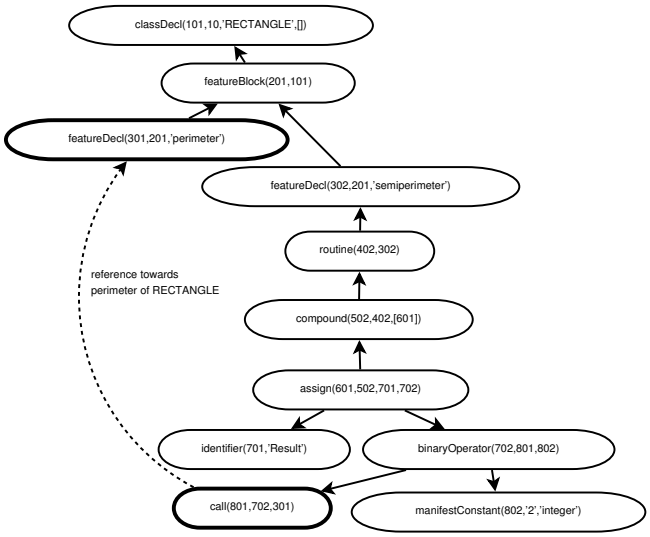

Figure 15: Local Calls Search

[3] presents such an approach that uses JTransformer ([12]) as the provider of logic facts, and is therefore stuck to using Java as the underlying programming language.

The design pattern under analysis is the Composite design pattern. Figure 18 shows the desired UML structure of a Composite anti-pattern.

The main problem with this structure is that the inheritance link between classes Composite and Shape is missing. Detecting such a problem is not as trivial as it seems to be. To fully mark this UML structure as a potential (though incomplete) Composite implementation, one should find a few additional elements, some of which are not visible from the UML diagram:

- the Shape class must have at least one descendant and must share at least one method with this descendant
- the Composite class must contain an array of Shape objects
- the Composite class must have at least one method that iterates through the array of Shape objects, calling the method that the Shape hierarchy shares

In figure 18 the Shape hierarchy is sound (there is at least one descendant of class Shape) and there is at least one method shared throughout the hierarchy (method area()). The Composite class must still provide at least a method that iterates the collection of Shape objects calling the area() method for each of them. If such a method doesn't exist then there is no

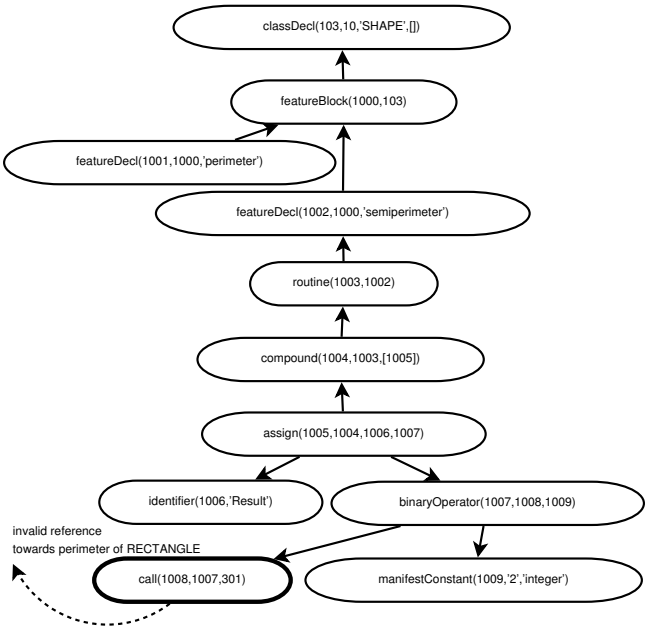

Figure 16: Method Body Clone

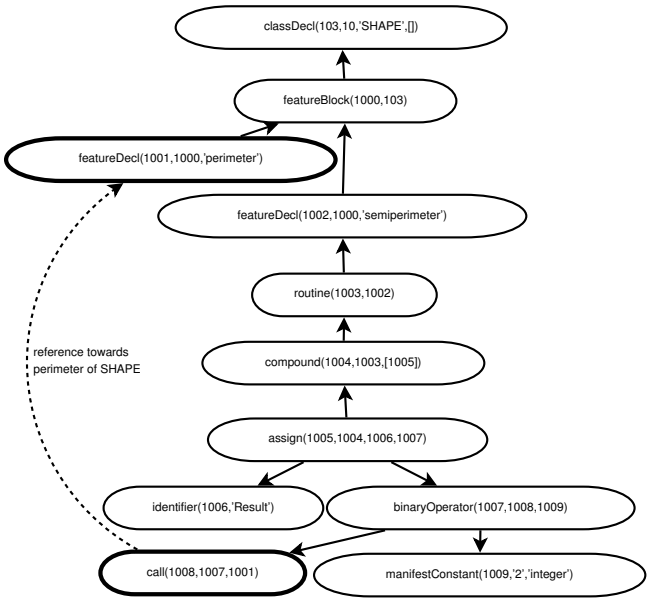

Figure 17: Local Calls Replacement

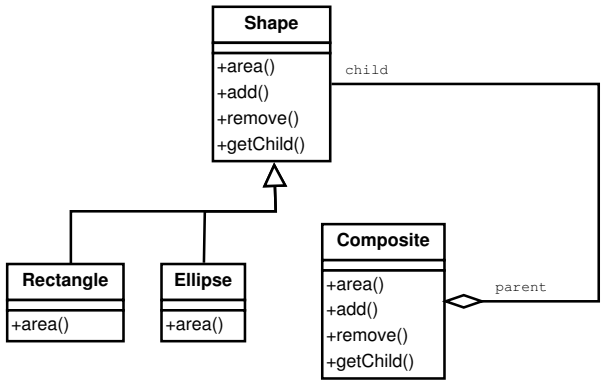

Figure 18: A Composite Anti-Pattern

reason for the UML structure to be modified to a Composite design pattern, since the advantages of a Composite are never used or needed. But if such a method existed, then making the Composite class a descendant of Shape would not alter the code behavior while at the same time would enhance it with the possibility to nest Composite instances indefinitely into one another, thus adding a certain elegance to the process.

[3] presents the methodology for such a detection on Java programs, using the logic based representation provided by the JTransformer tool ([12]). The use of a language independent logic based representation such as the one that ProGen would generate will have a very important consequence. If the mapping rules for several object-oriented language will be written such as similar entities will generate similar Prolog facts, then the detection technique described in [3] which is available for Java programs could also work without modifications on any other object-oriented language. Once at the Prolog level, there would ideally be no information about the programming language that was used to generate the logic based representation, so everything that is built on top of the Prolog representation (such as a Composite antipattern detection technique) would automatically become langauge independent, too.

7 Related Works

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

Stratego/XT [2] 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 [7] 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 mod-

els 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 [15] 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.

8 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.

We describe two concrete use cases for the logic representation. One use case describes the implementation of one exheritace class relationship mechanism. The generic routines dealing with searching, cloning, replacing nodes can be applied to any model which is described by the given metamodel. This means that those rules can be applied to programs written in any programming language, but expressed as logic facts respecting the metamodel. The metamodel presented in section 3 is not the only or the best choice to describe the structure of logic facts. The second use case deals with the detection of design anti-patterns in object-oriented systems. Anti-pattern detection Prolog rules are much expressive and close to natural language principle than writing imperative commands like visitors on an abstract syntax tree.

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 6 is complete.

References:

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, 3rd edition, USA, 2000.

[2] 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.

[3] Ciprian-Bogdan Chirila, Călin Jebelean, and Vladimir Creţu. A logic based approach to locate composite refactoring opportunities in object-oriented code. In *Proceedings of 2010 International Conference on Automation, Quality and Testing, Robotics - AQTR'2010*, volume 3, pages 130–136, May 2010.

[4] Ciprian-Bogdan Chirila, Călin Jebelean, Günter Kniesel, and Philippe Lahire. Generic rules for logic representation transformations. In *Proceedings of 2010 International Conference on Automation, Quality and Testing, Robotics - AQTR'2010*, volume 3, pages 142–148, May 2010.

[5] Free Software Foundation. SWI-Prolog, 2008.

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

[7] 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.

[8] ECMA International. Standard ECMA-367 Eiffel: Analysis, design and programming language. www.ecma-international.org, June 2006.

[9] 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.

[10] 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.

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

[12] 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.

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

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

[15] 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.

[16] Object Management Group. UML Superstructure version 2.0. www.omg.org/uml, October 2004.

[17] 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.