Generic Rules for Logic Representation Transformations

Ciprian-Bogdan Chirila[1], Călin Jebelean[1], Günter Kniesel[2], Philippe Lahire[3]
[1] Automation and Computer Science Faculty, University Politehnica of Timisoara, Romania,
E-mail: {chirila,calin}@cs.upt.ro
[2] Computer Science Department III, University of Bonn, Germany,
E-mail: gk@cs.uni-bonn.de
[3] Computer Sciences Department, Faculty of Sciences, University of Nice, France,
E-mail: philippe.lahire@unice.fr

***Abstract*—Programs expressed using logic representations can be more easily analysed and transformed. Transformations will depend on the target language semantics. A field encapsulation refactorization will be different for a Java program and an Eiffel program. Logic based representations of programs and its metamodel allows writing generic rules capable of performing some language independent transformations like: syntactical and semantical checking, searching specific nodes, cloning node structures, replacing references, generating fact visualizations. An example is given in this sense related to the implementation of feature body exheritance mechanism of reverse inheritance class relationship in the context of Eiffel language.**

I. Introduction

Modeling programs by logic facts brings several benefits. Logic represented programs can be analysed or modified by queries written in declarative languages in a more expressive manner than is permitted by imperative languages.

In [9] it is showed that any program represented by its grammar can be transformed into logic representation. In the context of logic representation, metaprogramming can help writing language independent rules for tasks like: type checking, subtree cloning, specific node searching, factbase visualization.

In this paper we present several generic rules for language independent program analysis [8] and transformation [13]. We will show that such generic rules can help in program analysis like anti-design pattern detection and in the implementation of a new class reuse relationship.

The paper is structured as follows. In section two we present the metamodel structure for the logic based representation. The third section presents the generic rules and what kind of transformation they do. In the fourth section we describe the generic rules we designed to help program transformation. In the fifth section related works are presented. Finally, in section six we conclude and we set the perspectives.

II. Metamodel

In this section we present the metamodel [10] of the factbase representation for programs.

A. Node Structure

In figure 1 we present the structure of an AST node and a relation node in the context of logic representation. To be more precise, we present the structure of a class declaration and the deferred relation from the Eiffel programming language [11], [7]. In Eiffel a deferred class is an abstract class as it is known in Java [3].

```
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).
```

Fig. 1. Metamodel Example

In logic representation a class declaration is defined as a fact having 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 - refers to the name of the field;

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

The 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 the AST node. In the context of this work the notion of fact and node and considered to be synonym. The name of the fact can be considered its type. Each argument is described by a *ast_arg* fact and has its own 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 ot 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 dedicated 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.

B. Node Arguments

The fact arguments are very important to our generic routines since our approach is based on them. 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.

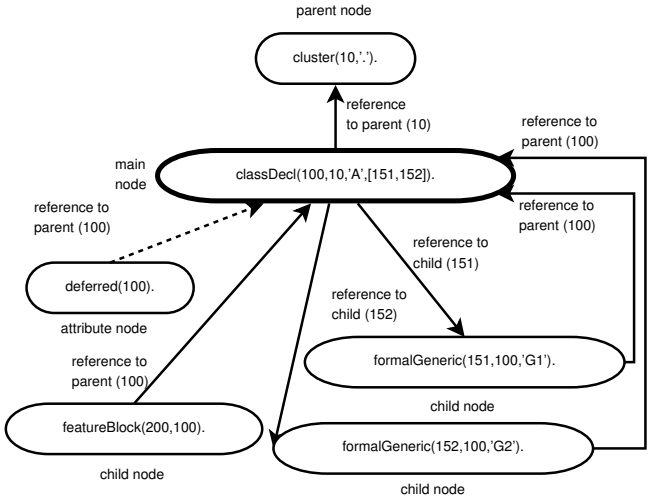

Fig. 2. AST Node Structure

In figure 2 we present an example of a class declaration node and its potential relations with other nodes from the tree structure:

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*;

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.

III. Generic Rules for Factbase Transformation

In the following subsections we will present several factbase transformation generic rules.

A. The Generic Algorithm Template

First, we will describe the main principles behind each generic algorithm. A generic algorithm gets as argument the identifier of the analysed structure root node. Because the algorithm is recurrent we will refer the analysed node as *current node*. Next, we describe the main steps:

(1) each node from the factbase is searched and decomposed into a list containing the type and the arguments of the fact. For example *classDecl(100,10,'A',[])* fact becomes *[classDecl,100,10,'A',[]]*. The second element in the list is the node id (*100*) while the first is the node type (*classDecl*). When the fact is found the fact structure information is retrieved from the metamodel and it is launched the arguments analysis.

(1a) if the argument is its own fact identifier some action may be executed;

(1b) if the argument is a parent identifier some action may be executed;

(1c) if the argument is a child identifier some action may be executed and step (1) may be called recursively on the child node;

(1d) if the argument is a child identifier list some action may be executed and step (1) may be called recursively on each child node;

(1e) if the argument is a reference identifier some action may be executed;

(1f) if the argument is an atom some action may be executed;

(2) we iterate the factbase in order to locate all children nodes referring as parent the current node and step (1) may be called recursively on each child node;

(3) we iterate the factbase to locate all relation nodes pointing to the current node and on each node actions may be executed.

B. Syntactical and Semantical Generic Checking

Syntactical and semantical checking are vital checks after model transformations in order to assure model integrity. This kind of checking verifies if all relations between facts respect the constraints of the metamodel. Syntactical and semantical checking works according to the following actions.

The checking algorithm follows the template presented in subsection III-A, but the actions are the following:

(1a) for own fact identifier no action is taken;

(1b) for the parent identifier we check if a fact with the given identifier and type (extracted from the metamodel) exists;

(1c) for the child identifier we check if a fact with the given identifier and type (extracted from the metamodel) exists;

(1d) for the child identifier list we check if fact with the given identifiers and type (extracted from the metamodel) exist;

(1e) for the reference identifier we check if a fact with the given identifier and type (extracted from the metamodel) exists;

(1f) for the atom arguments we check if the argument value is a Prolog atom;

(2) we call the checking algorithm on each child node;

(3) for each relation node we check if a fact with the given parent identifier and type (extracted from the metamodel) exists.

To illustrate how syntactical and semantical checking works in practice we will take the example (see figure 3) and its equivalent logic model (see figure 4) and we will present all the necessary tests step by step.

```
class A[G1,G2]
 feature
  m(p1,p2:INTEGER) is do end
end
```

Fig. 3. Syntactical and Semantical Checking (Eiffel Code)

First, let us present an example of a simple Eiffel class subject for checking. The class has two formal generic arguments, one feature block, one method with two parameters and an empty body.

```
01 cluster(10,'.').
02 classDecl(100,10,'A',[151,152]).
03 formalGeneric(151,100,'G1').
04 formalGeneric(152,100,'G2').
05 featureBlock(200,100).
06 featureDecl(300,200,'m').
07 formalArguments(400,300,[501,502]).
08 formalArgument(501,400,'p1',600).
09 formalArgument(502,400,'p2',600).
10 type(600,610).
11 classType(610,700).
12 classDecl(700,10,'INTEGER',[]).
```

Fig. 4. Syntactical and Semantical Checking (Prolog Factbase)

The example from figure 3 is translated in the Prolog factbase depicted in figure 4.

Next, we will present how our earlier described algorithm works on the factbase of figure 4.

Step (1)

We intend to check class *A*. Class *A* is denoted by *classDecl(100,10,'A',[151,152])* fact. The first argument is *100*, which represents the class own idenfier, information retrieved from the metamodel (see figure 1). For this argument no action is taken as we stated in step (1a) of the algorithm. Next argument, identifier *10* represents a reference to the class parent. From the metamodel we find out that it is a cluster node. At this point we check if there is in the factbase a *cluster* fact having identifier *10*. As we can see in the factbase, this verification checks. Further we have an atom, which needs no actions to be taken. Next, we have a list of two identifiers *151* and *152*. From the metamodel we get the information that they should represent formal generic facts. Interrogating the factbase by searching for formal generic facts with the given identifiers, the response is positive. Now, each formal generic node is checked recursively. This implies decomposing the node, interrogating its meta-information and analyzing its arguments in the same manner.

Step (2)

After analyzing the node and its arguments we proceed in finding all nodes reffering it. To be more accurate, we search for any fact having own identifier and having *100* as parent. The following nodes will result after the search: *featureBlock(200,100)*, *formalGeneric(151,100,'G1')* and *formalGeneric(152,100,'G2')*. These nodes will be verified recursively starting at step (1).

Step (3)

Finally, we search for all relation nodes having no own identifier, but only reference to the parent. In our example only one such node exists and it is the *deferred* node. We check according to its metamodel description if the type of its reference is a *classDecl*.

C. AST Node Subtree Generic Search

Searching AST nodes globally by type or by identifier is quite a simple task which can be carried out by the Prolog term decomposition operator *ASTNode =.. [Functor, Id | NodeArgumentIdList]*. But locating nodes by type within a given subtree is a more complicated task. There are two searching solutions:

i) top-down - by exploring the structure from the root to the leaf nodes;

ii) bottom-up - by assembling the global list using the decomposition operator and selecting only the nodes which have the given ancestor.

In our work we implemented the top-down searching approach. The algorithm works following the very same template presented earlier:

(1) we locate the node with the given identifier and we check its type, if it is equal to the searched one, the current fact is stored in a result list;

(1a) for own fact identifier no action is performed;

(1b) for the parent identifier no action is performed;

(1c) for the child identifier we call the search recursively;

(1d) for the child identifier list we call the search recursively on each child;

(1e) for the reference identifier we check its type, if it is equal to the searched one, the current fact is stored in a result list;

(1f) for the atom arguments no action is performed;

(2) we call the searching algorithm on each child node;

(3) each relation node's type is compared to the searched type, if it is equal to the searched one, the current fact is stored in a result list.

D. Subtree Generic Cloning

Cloning a subtree generically can help in situations where parts of programs are adapted or evolved. The goal of the algorithm is to create a copy of the source structure. The algorithm respects the earlier presented template and uses the following actions:

(1) we locate the node with the given identifier;

(1a) for own fact identifier we return a new generated identifier;

(1b) for the parent identifier we return the node identifier where the new structure will be located, usually it is given through a parameter;

(1c) for the child identifier we call the cloning node recursively and we return its result identifier;

(1d) for the child identifier list we call for each node the cloning rule recursively and we return the cloned nodes identifiers list;

(1e) for the reference identifier we return the very same reference;

(1f) for the atom argument we return the very same value of the atom;

(1) in this point we assemble the new fact from the cloned arguments;

(2) we call the cloning algorithm on each child node;

(3) for each relation node we clone it and link it to the clone of the current node.

Generic subtree deletion belongs to the same category of generic rules. Such rules may be useful also in program transformations.

An example illustrating how this algorithm works is presented in section IV.

E. Reference Generic Replacement

The proposed algorithm is useful when a node structure needs to be adapted to a different context. Usually, the structure is kept as such and references are replaced according to the new context using a map filled with the old and new references. The algorithm template actions are the followings:

(1) we locate the node with the given identifier;

(1a) for own fact identifier no action is performed;

(1b) for the parent identifier no action is performed;

(1c) for the child identifier we call the replacing rule recursively;

(1d) for the child identifier list we call for each node the replacing rule recursively;

(1e) for the reference identifier we replace it with its counterpart if it is present in the map;

(1f) for the atom argument no action is taken;

(2) we locate all children referring current node and we call the replacing rule recursively on each child node;

(3) for each relation node we analyze and replace its arguments from the map.

An example illustrating how this algorithm works is presented in section IV.

F. Generic Visualizations

These rules are able to provide human readable representations for facts in transformation debug contexts. One proposed representation is Prolog fact listing with tabular indentation which increases dramatically the readability of facts, especially for large factbases.

```
cluster(10,'.').
  classDecl(100,10,'A',[151,152]).
    formalGeneric(151,100,'G1').
    formalGeneric(152,100,'G2').
    featureBlock(200,100).
      featureDecl(300,200,'m').
        formalArguments(400,300,[501,502]).
          formalArgument(501,400,'p1',600).
          formalArgument(502,400,'p2',600).
            type(600,610).
              classType(610,700).
   classDecl(700,10,'INTEGER',[]).
```

Fig. 5. Indented Factbase Visualization Example

In figure 5 we present such a fact listing. The generic rules which implemented the listing have one basic principle: printing the facts with a computed identiation.

Another useful representation is based on XML. Even large factbases can be easily navigated and inspected using XML browsers.

```
<?xml version="1.0" encoding="UTF-8" ?>
<project>
<fact factname="cluster" id="10"
 clusterName=".">
  <fact factname="classDecl" id="100"
   cluster="10" name="A"
   formalGenerics="[151,152]">
    <fact factname="formalGeneric" id="151"
     class="100" name="G1"></fact>
    <fact factname="formalGeneric" id="152"
     class="100" name="G2"></fact>
    <fact factname="featureBlock" id="200"
     class="100">
      <fact factname="featureDecl" id="300"
       featureBlock="200" name="m">
        ...
      </fact>
    </fact>
  </fact>
</fact>
</project>
```

Fig. 6. XML Factbase Visualization Example

In figure 6 a fragment of such a generation is listed.

IV. Case Study: Implementing Method Body Exheritance for Eiffel Reverse Inheritance

Informally, reverse inheritance (exheritance) [13] is an inheritance class relationship where the subclasses exist first and the superclass is created afterwards. Reverse inheritance implements the generalization class relationship of UML [12]. On the other hand reverse inheritance is a class reuse mechanism which has several capabilities of:

i) allowing a more natural class hierarchy design;

ii) feature factorization from existing classes;

iii) reusing behavior from a class;

iv) extending a class hierarchy;

v) decomposing and recomposing classes;

vi) adding a new layer of abstraction in an existing hierarchy;

vii) favoring the use of design patters [6].

Eiffel [7], [11] programming language has several characteristics 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 features.

As mentioned earlier, one of the goals of this class relationship is to factor common features from existing subclasses and to create a new representant 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.

```
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
```

Fig. 7. Method Body Exheritance Example (Eiffel Code)

In figure 7 we present two existing classes *RECTANGLE* and *ELLIPSE* and a new class *SHAPE* created by reverse inheritance. 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*.

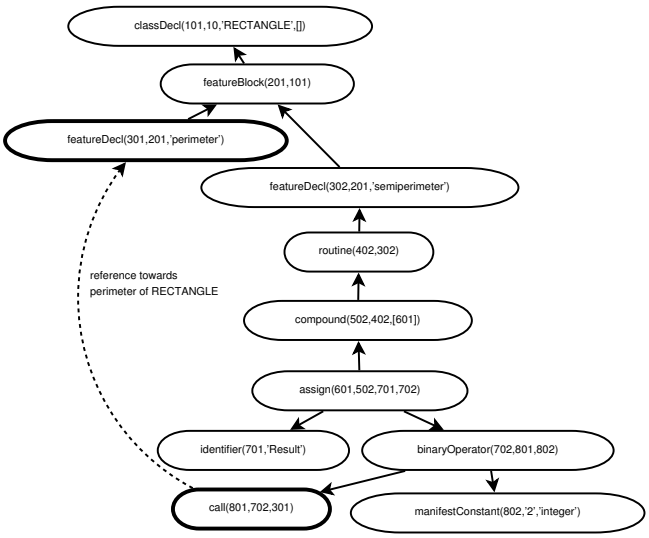

Fig. 8. Local Calls Search

In figure 8 we present the result of the local calls search. Using the **generic search rules** 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.

In figure 9 we duplicated the nodes of the method body in class *SHAPE* using the **generic cloning rules**. 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 10 we correct all invalid references. Using the **generic replacement rules** and the feature correspondence map from the exheritance process we replace all the invalid ex-local references with the correct ones.

V. Related Works

Program transformation techniques are used in a many areas of software engineering: program synthesis, optimization, refactoring, reverse engineering and documentation [1].

Using generated parsers, AST tree builders and implementing visitors [6] program transformation can be achieved. In such frameworks transformation rules are expressed in an imperative manner being less expressive than using the declarative approach. It is more natural to express a program

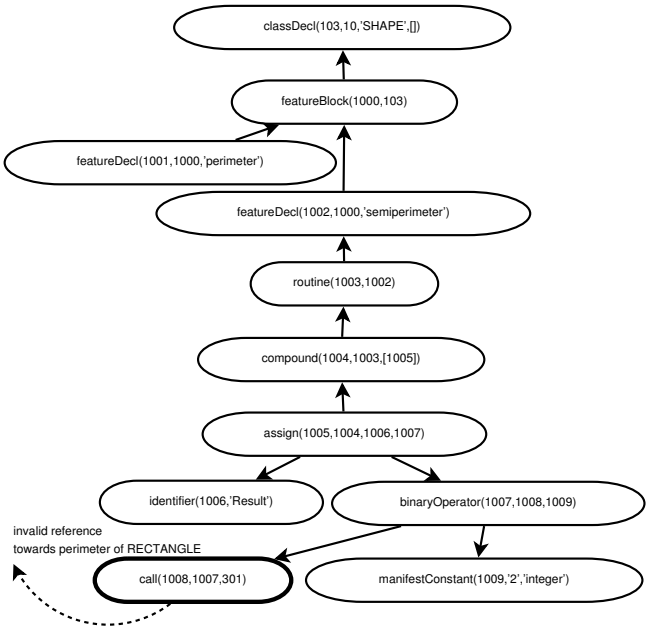

Fig. 9. Method Body Clone

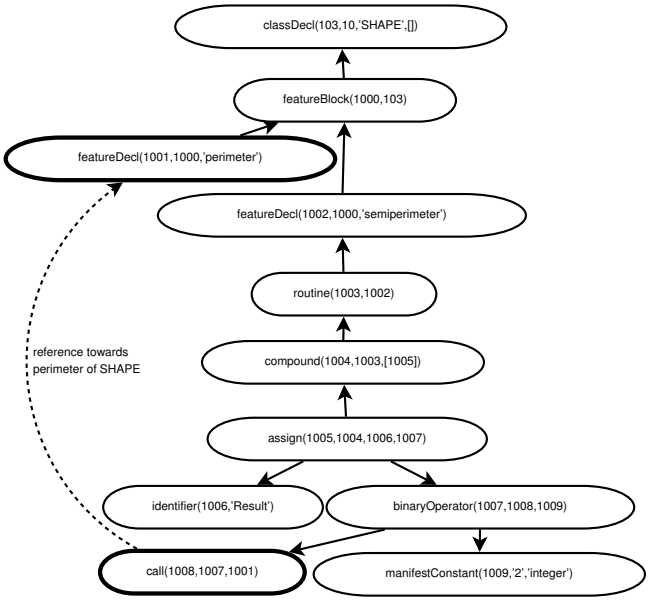

Fig. 10. Local Calls Replacement

transformation at conceptual level using the expressiveness of Prolog than writing the methods of a Visitor design pattern. On the other hand in order to benefit from genericity the imperative language must have generic capabilities. For example in Java [3] there is a meta-programming facility known as reflection.

Our generic rules provide compiler functionalities [2], like syntactical and semantical checking or pretty printing. These functionalities are not within our main goals, they came implicitly from the metamodel description and they prove to be crucial in testing the consistency of model represented programs.

In [4] are presented refactoring techniques in order to improve the design of the code.

We consider that using logic based representation respecting a metamodel and the generic rules some code transformations can be done more easily than with other formalisms.

VI. Conclusions and Perspectives

In this paper we showed that generic rules can be written for the analysis and transformation of logic representation described by a metamodel. We presented the implementation descriptions for these rules. Finally, we tackled about how these rules can help in the implementation of a new class reuse mechanism. We presented an Eiffel example of code reengineering method implementation reuse.

Generic rules expressiveness and their multi language capability are based on the metamodel. Generic rules help concrete rules in achieving the goal of program transformations.

The main drawback of generic rules is that they are time consuming. In our experiments on a factbase of 27 MB of Prolog facts, a cloning generic rule takes 3-4 minutes to clone an Eiffel method on a usual personal computer. This drives us to the conclusion that such rules are suitable to prototyping.

Another drawback is related to the complexity added to the program transformations by the generic rules. This however is balanced by the nature of the Prolog language itself which treats uniformly the meta and concrete levels.

In order to ameliorate generic rules time consumption there could be generated automatically concrete rules (checking, cloning, ...) for each programming language separately. Since the execution trace of the generic rules follows closely the metamodel, concrete rules may be generated for each metamodel entity.

References

[1] http://www.program-transformation.org.
[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison Wesley, 1986.
[3] K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, 3rd edition, USA, 2000.
[4] Martin Fowler. *Refactoring*. Addison-Wesley, March 2000.
[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] ECMA International. Standard ECMA-367 Eiffel: Analysis, design and programming language. www.ecma-international.org, June 2006.
[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] 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.
[10] Günter Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006.
[11] Bertrand Meyer. Eiffel: The language. http://www.inf.ethz.ch/˜meyer/, September 2002.
[12] Object Management Group. UML Superstructure version 2.0. www.omg.org/uml, October 2004.
[13] 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.