Towards Programs Logic Based Representation Driven by Grammar and Conforming to a Metamodel

Ciprian-Bogdan Chirila* and Călin Jebelean*
*University Politehnica of Timişoara, Romania
Faculty of Automation and Computer Science
Email: {chirila,calin}@cs.upt.ro

***Abstract*—Logic representation of programs gives an expressive way to perform analysis and transformation. Logic representation conforming to a specified metamodel enables analysis and transformation at both meta and concrete representation levels. Logic representation mapping rules express how programs can be automatically translated into metamodel conforming logic representation. The designed formalisms are suitable to any programming language.**

I. Introduction

Program logic representation [14] is used for analysis and transformation. Using logic representation several program analysis can be performed like: anti-pattern detection [13], concern detection and extraction [10]. Among the program transformations based on logic representation we can mention: refactorings, implementation of a reverse inheritance class relationship in Eiffel [16], [8]. On the other hand, metamodel conforming logic representation can be exploited by generic rules like: generic node search, generic type checking, generic node structure cloning.

In this paper we present how programs can be translated into a metamodel conforming logic representation driven by grammar (see figure 1). Using our approach we translate the

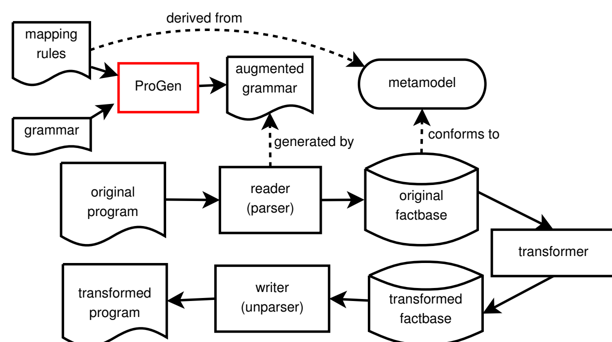

Fig. 1. Logic Based Approach

program source code into logic representation conforming to a chosen metamodel. In order to achieve our goal we use a translator (reader) which is built from a generated parser by a grammar and a set of mapping rules, representing semantical actions, which provides the logic representation according to the metamodel. Next, transformations may be performed on the original factbase resulting a transformed factbase. Using another translator (writer) the transformed factbase is regenerated back into the original program representation.

In [9], [6] we showed how concrete syntax tree equivalent logic representation can be obtained automatically, while now we provide a way to obtain abstract syntax tree equivalent logic representation. In order to do this we need the metamodel and mapping rules to specify how concrete syntax is abstracted into logic facts.

The paper is structured as follows. Section II shows how programs are modeled using logic based representation. In section III we present the metamodel of the logic program representation. In section IV we present the formalisms of the designed mapping rules. In section V we study how mapping rules are written for the MiniEiffel language. In section VI we present the related works, while in section VII we draw the conclusions and we set the future work.

II. Logic Based Representation of Programs

In order to show how programs can be represented using logical facts we will take an example of a class in an Eiffel [12] subset language, named *MiniEiffel*. This language subset is restricted to class declarations, feature blocks, feature declarations having formal arguments and type marks, instructions like: creations, assignments, calls and expressions. We consider these language features to be the most relevant from the modeling point of view. In both Eiffel and MiniEiffel class members (attributes and methods) are named feature declarations. These feature declarations can be grouped in feature blocks in the structural context of a class. In figure 2 we

```
01 class RECTANGLE
02 feature
03  width: REAL
04  height: REAL
05  make(w:REAL; h:REAL) is
06  do
07   width:=w
08   heigth:=h
09  end
10 end
```

Fig. 2. Rectangle Class Source Code

present a simple class modeling a rectangle (line *01*) having:

the *width* and *height* attributes (lines *03-04*) and a constructor method named *make* (lines *05-09*). This method takes as input the width and height of the rectangle as formal arguments and sets their values to the corresponding class members (lines *07-08*).

```
00 Cluster ::= (ClassDecl)*
01 ClassDecl ::= "class" <id> (FeatureBlock)* "end"
02 FeatureBlock ::= "feature" (FeatureDecl)*
03 FeatureDecl ::= <id> ["(" FormalArguments ")"] [":" <id>]
04   ["is" ("do" Routine | "deferred") "end"]
05 FormalArguments ::= FormalArgument (";" FormalArgument)*
06 FormalArgument ::= <id> ":" <id>
07 Routine ::= (Instruction)*
08 Instruction ::= Creation | Assignment | Call
09 Creation ::= "create" <id>
10 Assignment ::= <id> ":=" Expression
11 Expression ::= Call (<BinaryOperator> Call)*
12 Call ::= <id> [Actuals] ("." <id> [Actuals])*
13 Actuals ::= "(" Actual ("," Actual)* ")"
14 Actual ::= Expression
```

Fig. 3. MiniEiffel Grammar

In figure 3 we present the grammar of the *MiniEiffel* programming language to which our *RECTANGLE* class (see figure 2) conforms. The grammar describes a class declaration having tokens: ”class”, ”end” as keywords, *id* as identifier, representing the class name, and a list of feature blocks (see line *01*). A feature block is a list of feature declarations or class members (see line *02*). A feature declaration is described, in order, by:

i) an identifier *id* - the name of the feature;
ii) an optional list of formal arguments enclosed by ”(” ”)”;
iii) a return type expressed by the ”:” *id* sequence, the identifier denotes the type;
iv) the ”is” keyword;
v) a ”do” *routine* sequence if the feature is concrete or ”deferred” if the feature is abstract;
vi) the ”end” keyword (see lines *03-04*).

The routine is defined as a list of instructions (see line *07*). An instruction can be an object creation instruction, an assignment or a call (see line *08*). The creation instruction is represented by the ”create” keyword and the identifier of the feature to be created (see line *09*). The assignment instruction is denoted by the identifier to be assigned and the assigned expression (see line *10*). The expression is either a simple call or a successive application of a binary operator on two calls (see line *11*). A call is a qualified chain of identifiers optionally having actuals (see line *12*). The actuals are a list of actual elements separated by the ”;” token (see line *13*). Finally, in line *14* we learn that an actual is denoted by an expression.

Using a parser generator taking as input the previously listed grammar we obtain the MiniEiffel parser. Using this parser to analyze the *RECTANGLE* class we obtain its the concrete syntax tree.

In figure 4 we present the concrete syntax tree of the class listed in figure 2. The concrete syntax is represented by a tree composed of non-terminal nodes, while the terminals displayed as included in the non-terminals. For example, a *MiniEiffel* class will be represented by a node containing

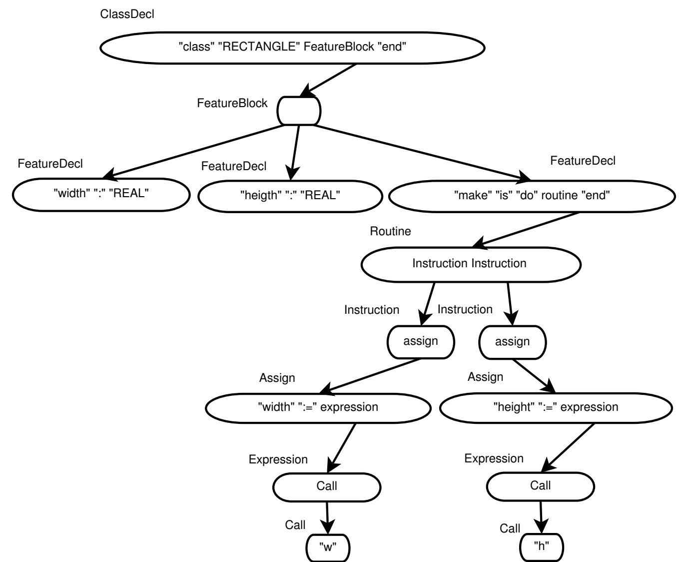

Fig. 4. Rectangle Class Concrete Syntax Tree

in order: ”class”, ”RECTANGLE”, *FeatureBlock*, ”end”. The first, second and fourth items are terminals, while the third one is a sub node representing a feature block.

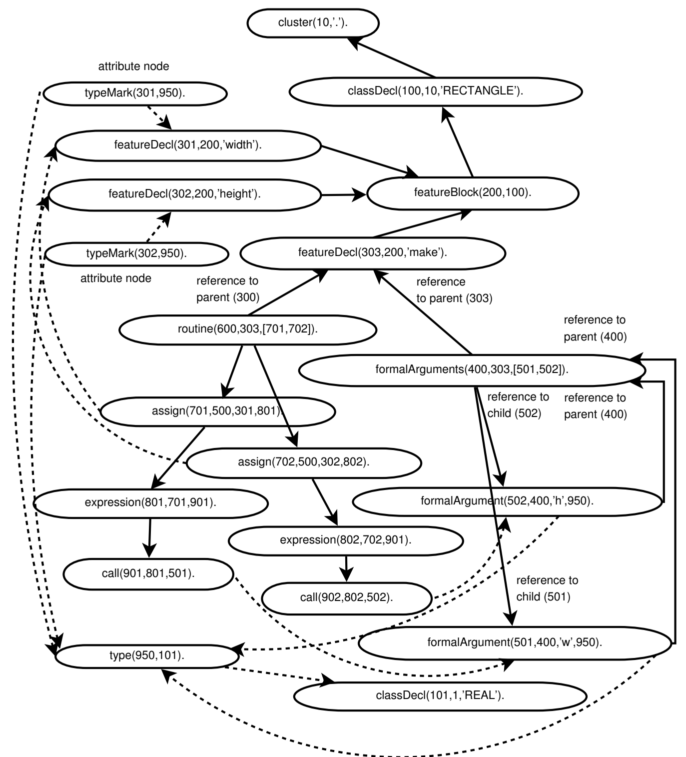

Fig. 5. Rectangle Class Logic Representation

In figure 5 we present the equivalent Prolog facts for the *RECTANGLE* class listed in figure 2. The class declaration is modeled by the *classDecl* fact which has the following arguments:

i) globally unique identifier (*100*);
ii) parent identifier - the cluster identifier the class belongs to (*10*);
iii) class name (*RECTANGLE*).

The feature block is a class member container and it is modeled as a child of the class declaration fact. The *featureBlock* fact has the following arguments:
i) globally unique identifier (*200*);
ii) parent identifier - the class identifier the feature block belongs to (*100*).

From the listing of the class code one can notice that three class members are defined: two attributes (*width*, *height*) and one method (*make*). Attribute *width* is modeled by the *featureDecl* fact having the following arguments:
i) globally unique identifier (*301*);
ii) parent identifier - the feature block identifier the attribute belongs to (*200*);
iii) attribute name (*width*).

A fact named *typeMark* is attached to the *featureDecl* fact, denoting the type of the modeled attribute, which has the following arguments:
i) parent identifier - the owner attribute identifier (*301*);
ii) type identifier - the attribute *REAL* type identifier (*950*). Attribute *height* is modeled in the same manner.

Method *make* is modeled by its *featureDecl* Prolog fact. This method has a set of formal arguments and a routine modeled by *formalArguments* and respectively *routine* facts. The *formalArguments* fact has:
i) globally unique identifier (*400*);
ii) parent identifier - the *featureDecl* fact identifier (*303*);
iii) the ordered list of each formal argument identifier (*[501,502]*).

Each formal argument is modeled by a *formalArgument* fact having:
i) globally unique identifier (*501,502*);
ii) parent identifier - the *formalArguments* fact identifier (*400*);
iii) the name of the formal argument (*w,h*);
iv) the argument type identifier (*950*) denoting type *REAL*.

The *routine* fact models the ordered list of instructions and has the following arguments:
i) globally unique identifier (*600*);
ii) the identifier of the parent method denoted by a *featureDecl* fact (*303*);
iii) the ordered list of instruction identifiers (*[701,702]*).

The body of the make method has two assignment statements (*701,702*), having the following arguments:
i) globally unique identifier (*701,702*);
ii) parent identifier - the routine identifier (*600*);
iii) the assigned feature identifier (*301,302*) for *width* and respectively *height*; iv) the assigned expression identifier (*801,802*).

The two expressions (*801,802*) are modeled by the following arguments:
i) globally unique identifier (*801,802*);
ii) parent identifier - the assign identifier (*701,702*);
iii) the identifier of a feature call (*901,902*).

The two calls (*901,902*) are modeled by the following arguments:
i) globally unique identifier (*901,902*);
ii) parent identifier - the expression identifier (*801,802*);
iii) the identifier of the called feature (*501,502*) for *width* and *height*.

III. The Metamodel of the Logic Representation

In this section we will present a fragment from the *MiniEiffel* metamodel to which all the Prolog facts will conform. We restricted the metamodel description to the most relevant facts because of space reasons but still covering all possible cases. In figure 6 we list the metamodel rules for the Prolog facts

```
01 ast_node_def('MiniEiffel',featureDecl,[
02  ast_arg(id, mult(1,1,no), id, [featureDecl]),
03  ast_arg(parent, mult(1,1,no), id, [featureBlock]),
04  ast_arg(featureName, mult(1,1,no), attr, [atom])
05 ]).
06 ast_relation('MiniEiffel',deferred,[
07  ast_arg(featureDeclRef, mult(1,1,no), id, [featureDecl])
08 ]).
09 ast_node_def('MiniEiffel',formalArguments,[
10  ast_arg(id, mult(1,1,no), id, [formalArguments]),
11  ast_arg(parent, mult(1,1,no), id, [featureDecl]),
12  ast_arg(formalArgs, mult(1,*,ord), id, [formalArgument])
13 ]).
14 ast_node_def('MiniEiffel',formalArgument,[
15  ast_arg(id, mult(1,1,no), id, [formalArgument]),
16  ast_arg(parent, mult(1,1,no), id, [formalArguments]),
17  ast_arg(formalArgumentName, mult(1,1,no), attr, [atom]),
18  ast_arg(typeRef, mult(1,1,no), id, [type])
19 ]).
20 ast_sub_tree('MiniEiffel',formalArgs).
21 ast_ref_tree('MiniEiffel',featureDeclRef).
22 ast_ref_tree('MiniEiffel',typeRef).
23 ast_ancestor_tree('MiniEiffel',parent).
```

Fig. 6. Metamodel Fragment

listed in figure 5. Each metamodel rule describes an AST (Abstract Syntax Tree) node and the legal argument values for the corresponding facts.

The *featureDecl* AST node is defined by: i) the *ast_node_def* fact which has as first argument the name of the programming language the node refers to (*MiniEiffel*); ii) the name of the node (*featureDecl*), followed by a list of descriptive arguments.

In the context of this work the notions of fact and node are considered to be synonyms. The name of the fact can be considered its syntactical 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* (lines *02, 10,15*) or *parent* (lines *03,11,16*), but the rest can be freely chosen (lines *04,12,17,18*);
ii) multiplicity - can be one to one (lines *02,03,04,...*) or one to many (line *12*);
iii) ordering - it makes sense when multiplicity is one to many and in this case the argument is a list which can be ordered or not. For example a formal argument list may have one or more formal arguments and their order is an important syntactical information.
iv) kind of value - it can be identifier - *id* (lines *02,03*) or attribute - *attr* (line *04*). 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, for *featureDecl* facts: a) the type of the identifier argument is *featureDecl*; b) the type of the feature declaration parent is *featureBlock*; c) the type of the *featureName* argument is *atom*.

One can notice that in the metamodel there are rules for structural nodes expressed as *ast_node_def* facts and also for relations which are expressed as *ast_relation* facts. For example, *featureDecl*, *formalArguments*, *formalArgument* are AST nodes, while *deferredFeature* is a relation node.

The relation between AST nodes is determined in the metamodel by the type of the fact arguments. For example *formalArgs* in line *20* is declared as a subtree. This means that the relation between the *formalArguments* fact and *formalArgument* fact is a structural one, a sub-node relation. On the other hand, in lines *18,22* the *typeRef* argument represents a relation between the *formalArgument* fact and a *type* fact. Of course that the *parent* argument denotes an ancestor structural relation between nodes (see lines *03,11,16,23*).

IV. MAPPING RULES: FROM CONCRETE TO ABSTRACT SYNTAX

In this section we will present a set of simple rules which allow mapping concrete syntax to abstract logical facts designed on the framework of the JavaCC [15] parser generator library.

A. The JavaCC Library Extension

In order to facilitate the mapping of concrete syntax into logic facts we augmented the JavaCC [15] class library to have better access to the CST (Concrete Syntax Tree) information. The default CST token access mechanism is based on a few methods from *SimpleNode* generated class and is depicted in figure 7. In this figure we present a fragment of the

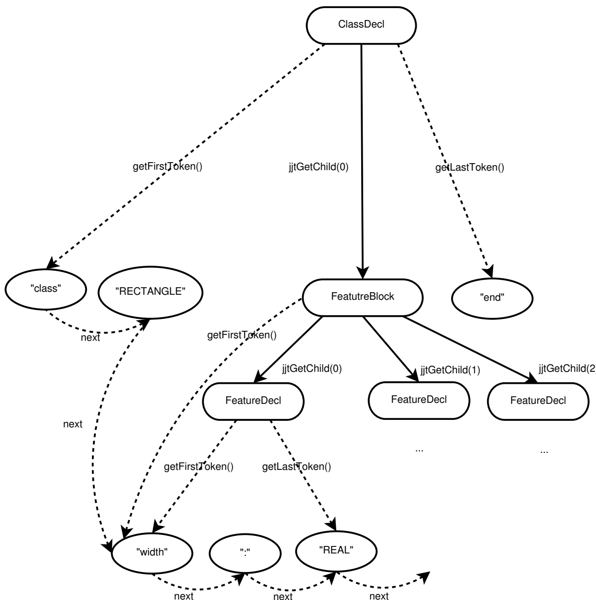

Fig. 7. JavaCC Node Navigation Methods

RECTANGLE class CST. The only available node access methods are limited to: i) children non-terminals (*jjtGetChild(int)* and *jjtGetNumChildren()*); ii) the first and the last terminal (*GetFirstToken()* and *GetLastToken()*). There is no possibility to access directly the first, second, third, etc child of a node regardless of its non-terminal or terminal kind. The tokens of a non-terminal can be navigated as a linked list, but the navigation is driven through the whole subtree frontier. On the other hand, the presence of a token can not be located by a name search.

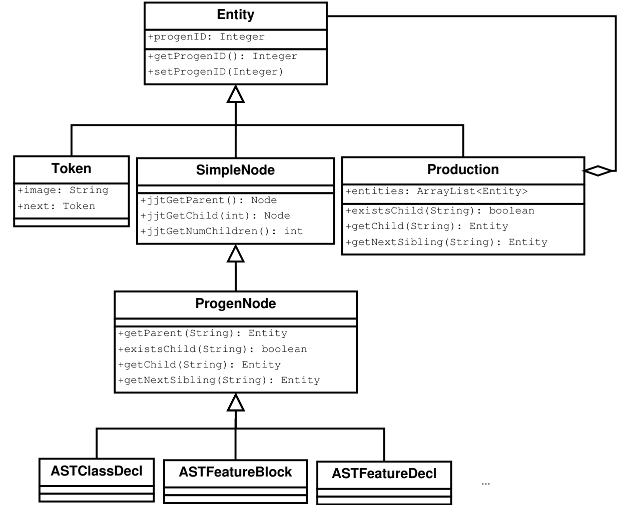

Fig. 8. JavaCC Library Extension

In figure 8 we add new classes to the library in order to simplify the access to the node children. In order to achieve this goal, firstly, we design a new *Entity* superclass for both *Token* and *SimpleNode* classes. We mention that these two classes are automatically created by the JavaCC parser generator and the two inheritance relationships must be crated manually or programmatically. Secondly, we create a new *Production* class as subclass of *Entity* which handles a collection of *Entity* instances, having methods to access homogeneously both tokens and non-terminals from the CST. Thirdly, a new *ProgenNode* class is created as subclass of *SimpleNode* and superclass for all the AST node generated classes named (*ASTClassDecl*, *ASTFeatureBlock*,...). This class will use all facilities from the *Production* class. To set the superclass of all generated AST node classes we use the *NODE_CLASS="ProgenNode"* JavaCC option. After parsing, a visitor will add to all nodes unique identifiers accessible through *getProgenID()* and *setProgenID()* methods inherited from class *ProgenNode*.

B. Mapping Formalisms

Mapping rules are expressed as simple formalisms which can be easily translated into semantical actions to be inserted in the grammar, in order to generate logic representation. The designed formalisms have two basic principles:
i) the location of a sub-node (terminal or nonterminal) in the

context of a node;
ii) the generation of a semantical action, which will be executed after the previously located sub-node.

```
01 LanguageGrammarRuleName (TokenConstantValue | Expression)
02   generate FactName(Expression1,Expression2,...)
03 LanguageGrammarRuleName
04   return Expression
```

Fig. 9. Mapping Rules Grammar

In figure 9 we present the formalisms for the two kinds of mapping rules used in logic facts generation. The formalisms between lines *01-02* denote rules that specify how a Prolog fact will be generated from the CST. The token constant value or the expression is located in the CST node and afterwards a Prolog fact will be generated having as arguments the values denoted by the listed expressions. The formalisms between lines *03-04* specify that some grammar rules, equivalent to methods in the JavaCC generated parser, will return certain values denoted by the listed expressions.

V. Case Study: Mapping Rules for MiniEiffel

In this section we will present what mapping rules are necessary to translate the CST representation (figure 4) of class *RECTANGLE* into equivalent logic representation (figure 5). For clarity reasons we split the mapping rules in two sets: one related to the class structure (figure 10) and the other related to the routine (figure 11).

```
01 Cluster node.jjtGetChild(0) generate
02   cluster(node.getProgenID(),
03   '.'
04   ).
05 ClassDeclaration "class" generate
06   classDecl(node.getProgenID(),
07     node.getParent().getProgenID(),
08     "'"+node.getNextSibling("class")+"'"
09   ).
10 FeatureBlock "feature" generate
11   featureBlock(node.getProgenID(),
12     node.getParent().getProgenID()
13   ).
14 FeatureDecl node.jjtGetChild(0) generate
15   featureDecl(node.getProgenID(),
16     node.getParent().getProgenID(),
17     "'"+node.jjtGetChild(0).toString()+"'"
18   ).
19 FormalArguments return
20   orderedList(FormalArgument)
21 FormalArguments node.jjtGetChild(0) generate
22   formalArguments(node.getProgenID(),
23     node.getParent().getProgenID(),
24     node.toString()
25   ).
26 FormalArgument node.jjtGetChild(0) generate
27   formalArgument(node.getProgenId(),
28     node.jjtGetParent().getPGId(),
29     "'"+node.jjtGetChild(0).toString()+"'",
30     -1
31   ).
```

Fig. 10. Class Mapping Rules

In figure 10 we present how logic facts are generically created from the CST according to the metamodel. Between lines *01-04* a *cluster* fact is generated. It has a unique identifier obtained from the current node by the *node.getProgenID()* call and a path which in our case is set to the current directory '.' constant. Between lines *05-09* a *classDecl* fact is generated after the "class" keyword is located in the context of the current node. The fact has a unique identifier got from the current node. It has a parent identifier obtained through the *node.getParent().getProgenID()* call of line *07*. The name of the class declaration is obtained by accessing the next sibling of the earlier located node and enclosing its String representation into apostrophes (line *08*). The *featureBlock* fact is generated after the "feature" keyword is located. Such a fact has only unique and parent identifiers computed as in the previous fact generation (lines *10-13*). In the context of an *ASTFeatureDecl* node a *featureDecl* fact is generated having unique and parent identifiers. To be noted that there is no special keyword to trigger the generation. Instead, the presence of the first child node is used as acceptance for generation. One can notice that in the CST the name of the feature is the first token of the node. The third argument is generated by accessing the String representation of that node and enclosing it by apostrophes (line *17*). The *ASTFormalArguments* node has two mapping rules: one for fact generation and the other for setting the returned value. The rule between lines *19-20* specifies that the *ASTFormalArguments* node String representation will be an ordered list of formal argument identifiers. Usually, such a specification is dedicated to lists. The rule listed between lines *21-25* is generating the *formalArguments* fact having the following values: the unique identifier, the parent identifier and the node String representation, namely the list of formal argument identifiers described earlier. The *ASTFormalArgument* node holds information needed for the generation of the *formalArgument* fact. The first two arguments are the unique identifier and parent identifier. The third argument represents the name of the formal argument accessible from the current node (line *29*). The fourth fact argument is the type identifier of the formal argument. This identifier represents a reference, whose computation is not covered in this paper. In this stage of the research we set its value to a temporarily invalid -1 constant (line *30*).

In figure 11 we list the mapping rules for the rest of the facts. They are written according to the same principles explained earlier.

VI. Related Works

All today's models are organized around EMF [1] while all metamodels around Ecore and MOF [11]. The EMF project is a modeling framework and Java code generation facility for building tools and other applications based on a structured data model. In our approach the base is set on logic representation and its metamodel, both expressed through Prolog facts.

The EMFText [3] project allows to describe syntax for languages described by an Ecore model. It offers both "readers" and "writers" for DSLs (Domain Specific Language). Our work offers a generic way to express what such a reader should generate as Prolog facts, being more oriented to model object-oriented programming languages.

```
01 Routine return
02   orderedList(Instruction)
03 Routine node.jjtGetChild(0) generate
04   routine(node.getProgenID(),
05     node.getParent().getProgenID(),
06     node.toString()
07   ).
08 Instruction return
09   node.jjtGetChild(0).getProgenID()
10 FeatureDecl "deferred" generate
11   deferred(node.getProgenID()
12   ).
13 Creation "create" generate
14   creation(node.getProgenID(),
15     node.getParent().getParent().getProgenID(),
16     -1
17   ).
18 Assignment ":=" generate
19   assign(node.getProgenID(),
20     node.getParent().getParent().getProgenID(),
21     -1,
22     node.getNextSybling(":=").getProgenID()
23   ).
24 Expression node.jjtGetChild(0) generate
25   expression(node.getProgenID(),
26     node.jjtGetChild(0).getProgenID()
27   ).
28 Call node.jjtGetChild(0) generate
29   call(node.getProgenID(),
30     node.getParent().getProgenID(),
31     -1
32   ).
33 Actual node.jjtGetChild(0) generate
34   actual(node.getProgenID(),
35     node.getParent().getParent().getProgenID(),
36     -1,
37     node.jjtGetChild(0).getProgenID()
38   ).
```

Fig. 11. Routine Mapping Rules

The Kermeta [4] workbench is a metaprogramming environment based on an object-oriented DSL optimized for metamodel engineering. Our work is similar to Sintaks project from the Kermeta workbench. They use a special language to express the concrete syntax while our approach is based on the language grammar.

VII. Conclusions and Future Work

In this work we showed how abstract program logic representation can be obtained without learning new technologies, but just writing generic expressions in the metamodel framework using the extended version of the JavaCC class library.

The approach based solely on concrete syntax tree depends only on the language grammar, while the one based on the abstract syntax tree needs a metamodel and a set of mapping rules between the concrete syntax and the abstract syntax, as a tradeoff.

The other alternative to generate logic representation is to write a language specific translators like JTransformer [10] for Java or ETransformer [8] for Eiffel.

In this stage of our research we allow expressing mostly subtree relations between AST nodes / facts, while the expression of the semantical relations is set as future work. We intend to offer the possibility to express generically the references to classes, features, types,... using only their names taking into account scoping and other language features. For the moment, the temporarily invalid reference values can be replaced with user defined function calls capable of performing computation according to the concrete language semantics.

An immediate perspective is to implement the approach as a software tool and to integrate it in the Eclipse [2] platform as a plugin.

In order to validate our approach, we intend to experiment it on the GOBO Eiffel grammar [7] and to generate the same Prolog facts as the ETransformer [8] Eiffel to Prolog translator.

References

[1] Eclipse modelling framework. http://www.eclipse.org/modeling/emf/.

[2] Eclipse project. http://www.eclipse.org.

[3] Emf text project. http://http://www.emftext.org.

[4] Sintaks - Bridging concrete and abstract syntax. http://www.kermeta.org/sintaks/.

[5] SWI prolog. http://www.swi-prolog.org.

[6] Călin Jebelean, Ciprian-Bogdan Chirila, and Anca Maduta. 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 28-30 2008.

[7] Eric Bezault. GOBO Eiffel Project. http://www.gobosoft.com, November 2007.

[8] Ciprian-Bogdan Chirila. *Generic Mechanisms to Extend Object-Oriented Languages. The Reverse Inheritance Class Relationship*. PhD thesis, University Politehnica of Timişoara, February 26 2010.

[9] Ciprian-Bogdan Chirila, Calin Jebelean, and Anca Maduta. Towards automatic generation and regeneration of logic representation for object-oriented programming languages. In *In Proceedings of International Conference on Technical Informatics - CONTI 2008*, volume 2, pages 13–18, Timisoara, Romania, June 5-6 2008. Politehnica Publishing House Timisoara.

[10] Tobias Rho Gnter Kniesel, Jan Hannemann. A comparison of logic-based infrastructures for concern detection and extraction. In *Workshop on Linking Aspect Technology and Evolution (LATE'07), in conjunction with Sixth International Conference on Aspect-Oriented Software Development (AOSD.07), March 12-16, 2007, Vancouver, British Columbia*. Workshop on Linking Aspect Technology and Evolution (LATE'07), in conjunction with Sixth International Conference on Aspect-Oriented Software Development (AOSD.07), March 12-16, 2007, Vancouver, British Columbia, Mar 2007.

[11] Object Management Group. Meta-object facility. http://www.omg.org/mof.

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

[13] 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.

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

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

[16] 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.