Personal Contributions Regarding Software Reusability Description of Reverse Inheritance Semantics Using Prolog

PhD Research Report #3

Author: asist. univ. ing. Ciprian-Bogdan Chirilă

PhD Supervisor: prof. dr. ing. Ioan Jurca
PhD Co-supervisor: prof. Philippe Lahire (University of Nice, France)

Faculty of Automation and Computer Science

University Politehnica of Timişoara

July 25, 2008

Contents

Abstract

The reverse inheritance semantics was defined fully in the previous research report, but it was not yet implemented in any programming language. To prove that reverse inheritance is a feasible class relationship that helps Eiffel class reusability, we built a prototype that implements it. The extended Eiffel language is modeled using Prolog facts thus generating factbases corresponding to object-oriented systems. The semantics of the reverse inheritance concept will be expressed by model transformations applied to the factbase. The factbase model is transformed using the mechanisms of conditional transformations, which can detect transformation dependencies or can find a possible order for the transformation execution. Finally, the resulted model facts are translated automatically into pure Eiffel compilable code, in order to build the executable object-oriented system.

Acknowledgements

The current research report is part of the PhD programme developed in the context of the collaboration between University Politehnica of Timisoara, Romania and University of Nice from Sophia-Antipolis, France, where I had several internships, the last one held in 2007 for two months.

I would like to thank professor Ioan Jurca for his great efforts in supervising my PhD research activity, sustaining my research projects and elaborating the reviews for the research reports.

I would like to thank professor Philippe Lahire for the financial and intellectual resources invested in the thesis research. Specially, I want to thank professor Michel Riveill for financing a working visit in Bonn.

I would like also to thank professor Markku Sakkinen from University of Jyväskylä, Finland, for showing points of interest for the developed problematics during one of my PhD internships and for helping with the development of the semantics.

I would like to express my gratitude to dr. Günther Kniesel from the Informatics Institute of Bonn, Germany, for the effort invested in the prototype implementation and for organizing a working visit at the University of Bonn, Germany.

I want to express my regards to Mathieu Acher and Jean Ledesma for releasing the first version of one of the prototype modules.

I want to thank also to the head of our department professor Vladimir Creţu for the punctual advices and management information he gave me during the research activity. I would like to thank also the dean of our faculty, professor Octavian Proştean for granting the financial support in the research internships.

Chapter 1

Introduction

In this report is presented the **operational semantics** of reverse inheritance in order to convince the reader about the feasibility of this class relationship. We will propose an implementation solution for the reverse inheritance concept in the context of Eiffel language upon the **formal semantics** described in [Chi08]. The implementation is a full solution which offers the programmer a way to reuse already existing classes by reverse inheritance and to compile the resulted object-oriented system into an executable binary. The prototype stands as a proof of a concept in order to demonstrate that the proposed class relationship is feasible and it can be used practically, on class hierarchies, facilitating their reuse in different contexts. The extended language will be referred to as **RIEiffel**.

1.1 Reverse Inheritance Class Reuse Mechanism

In [Chi06] some use cases reverse inheritance was designed for, are presented:

- designing class hierarchies in a more natural way - it is more natural to model the subclasses then to notice commonalities and factor them in superclasses;
- capturing common functionalities of classes - when several classes have features with the same semantics they could be factored in a common interface to get a homogeneous access;
- inserting a class into an existing hierarchy - when the design of a class hierarchy lacks a level of abstraction using reverse inheritance and ordinary inheritance, a new class can be added "in the middle";
- extending a class hierarchy reusing partial behavior of a class - using reverse inheritance some feature implementation can be exherited in the superclass to be later inherited in other subclasses;
- creating a new type - reverse inheritance can be used only to create new types, without exheriting any feature, in order to manipulate variables;
- decomposing and recomposing classes - using reverse inheritance, behavior can be extracted in one superclass and can be combined with behavior from another superclass in a common subclass of the two;
- late integration of classes into design patterns - for example, the implementation logic of a template method design pattern can be exherited in the foster class in order to be inherited later in another subclass, reimplementing the method calls.

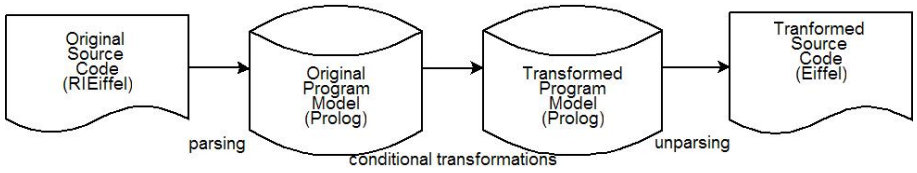

Figure 1.1: Generating Eiffel Source Code

1.2 Several Implementation Approaches

When adding an extension to an already existing programming language there are several implementation choices. In the next subsections we propose three choices and we analyse their advantages and drawbacks, motivating the choice selected for implementation.

1.2.1 Modifying the Eiffel Compiler

Apparently, a simple and straight forward idea is to modify the Eiffel compiler, including the new language extension and thus providing the executable of the object-oriented system. Such an approach implies writing the code expressing the semantics of reverse inheritance together with the rest of the compiler code. Thus the visibility of the reverse inheritance operational semantics would be diminished. On the other hand, it could be very difficult to prove that there are no deviations from the philosophy of the language and that illegal class constructions are not allowed. Also, such a task would be too big for a PhD research project. One more reason not to choose such an approach is that even most Eiffel compilers do not generate platform native binary code, but C or Java [AG00] sources instead.

1.2.2 Generating C Source Code

Another choice, which is used also by some Eiffel compilers, is to generate intermediary low level code which can be compiled afterwards by a platform independent compiler. For example ISE Eiffel [Sof08] compiler generates C code which is compiled further by gcc [SF08] on Unix/Linux [TOG08] or by Borland C [BI08] or Microsoft Visual C++ [Cor08b] on Microsoft Windows [Cor08a] platforms. Such an approach benefits from the fact that the resulted code is cross platform compilable, but would still lack in reverse inheritance semantics visibility.

1.2.3 Generating Eiffel Source Code Using Model Transformation

Our chosen implementation solution is based on the fact that class hierarchies built with reverse inheritance allow the same facilities as those built with ordinary inheritance, but in different orders of construction, and that the resulted class hierarchies have the same semantic properties. The main idea of the implementation is to translate class hierarchies having both reverse inheritance and ordinary inheritance into semantically equivalent classes having just ordinary inheritance. This solution involves (see figure 1.1): i) a translation from RIEiffel sources into Prolog model; ii) a transformation from Prolog facts to Prolog facts where the reverse inheritance facts are replaced with semantically equivalent ordinary inheritance facts; iii) a model translation into pure Eiffel source code compilable by an ordinary compiler. Thus the reverse inheritance semantics transformations are expressed using Prolog rules and the output is a pure Eiffel class hierarchy. Next, we define several notations related to the transformed items previously mentioned.

Original Source Code The **original source code** is the RIEiffel code which contains reverse inheritance and also ordinary inheritance. This code includes classes from different libraries which probably were developed in different contexts.

Original Prolog Model The **original Prolog model** is a factbase created by parsing the original source code. The original source code and the original Prolog model are semantically equivalent. Still some details like code indentation and organization are missing from the model, but these aspects do not change the logic of the original code. The model contains facts dealing with both reverse and ordinary inheritance. This model is subject for transformation.

Transformed Prolog Model The **transformed Prolog model** is the model obtained after transformations are applied on the original Prolog model. The transformed and original Prolog models are semantically equivalent. All reverse inheritance related facts are replaced with ordinary inheritance facts through transformations performed on the original Prolog model.

Transformed Source Code The **transformed source code** is obtain by reverse engineering from the transformed Prolog model. This code contains only ordinary inheritance and is compilable with an ordinary Eiffel compiler.

We have to state that this solution is just one implementation solution which requires the source code of the reused classes but in change offers the possibility of proving the equivalence of the two class hierarchies and thus the consistency of the reverse inheritance semantics.

1.3 Structure of the Report

In the second chapter we will present the reification of Eiffel and of the reverse inheritance extension in Prolog. We will discuss about how the model was build upon the grammar rules and what are the constraints a model must respect in order to be consistent. The third chapter will present the model transformations in Prolog that expresses the semantics of reverse inheritance in a formal manner. Some aspects are presented in the containing sections about feature exheritance, type exheritance, assertion composition, feature redefinition, feature adaptation, feature migration and exheritance facts removal. In chapter four we draw the conclusions and we set the future work.

Chapter 2

Eiffel Reverse Inheritance Reification in Prolog

In this chapter we will present the model of reverse inheritance class relationship. As the new class relationship semantics cannot be isolated from the rest of the language, we will have to model also the Eiffel language elements. In reverse inheritance we have a superclass, which is is build on the top of already existing subclasses. The superclass in reverse inheritance is called **foster class** and the subclasses are called **exherited classes** [Chi08]. The common features from the exherited classes can be selected to be exherited in the foster class. In order to be able to perform changes on the Eiffel code and to obtain a pure class hierarchy we need a model which can be changed easily. A Prolog fact base operated by conditional transformations represent a good choice for the implementation.

2.1 Reification of Eiffel

In order to integrate reverse inheritance class relationship in Eiffel programming language, first we augment its grammar (see appendix A) with the elements of reverse inheritance. For the implementation we chose the grammar from Eiffel GOBO library [Bez07]. Each grammar rule is analyzed and represented in the model by Prolog parameterised clauses. The factbase is designed in a similar way as a relational database is. Usually, each fact will have a unique key, which is represented by the first parameter of that fact. Facts having the same name may be considered belonging to the same table. The identifiers of the facts having same names represent the primary key of the table. Some facts denote entities which have parent entities. This type of relations are represented by identifiers which are second arguments in the clause and refer the parents. The links between the clauses are based on these identifiers.

Most of the rules refer to their parent rule. For example the relation between a class declaration and its cluster is modelled as done in subsection 2.1.1.

Some rules have navigation capabilities in both ways. For example, in subsection 2.1.5, when modelling formal arguments we need to store their order so for that we use a fact having a list as parameter. Also each formal argument is modeled by a fact which keeps a link to the parent list. So the list points to its children (formal arguments) as well as the children point to the parent list (formal argument list).

Some clauses have no own identifier at all since they denote some optional single attribute of another clause. For example the **deferred** keyword for a class will be modelled using an attribute clause which will refer to the feature and without any other information in subsection 2.1.5.

A special exception is represented by modelling expressions in subsection 2.1.8 where expressions use a downward referencing philosophy: each operator, unary or binary, will refer to its children. This decision was taken in order to reduce as much as possible the number of facts in the metamodel, thus decreasing its complexity.

Another remark must be noted about different facts which have to be handled in a uniform manner like instructions and expressions. For such kind of facts we have modeled a generic fact which refers the concrete facts. For example in subsection 2.1.7 the *instruction* generic fact will refer to all possible concrete facts modelling instructions like: *creation*, *assign*, *conditional*, *multi-Branch*, *loop*, *debug*, *check*. Thus, the *compound* fact which models an ordered set of instructions will refer only to *instruction* facts.

Before starting the description of the metamodel we must specify that all fact arguments beginning with "#" symbol represent identifiers, they behave like the indexes or keys in a database table. Actually, the resulting model is a set of Prolog clauses having the organization of a 3NF normalised [ref] relational database model without null keys.

2.1.1 Reification of Class Header

Project The first fact modelled is the one locating the Eiffel project. We are interested in the name of the project, its location along with the same information about the transformed project or output project:

```
project('ProjectName','Location','OutputProjectName','OutputProjectLocation').
```

This fact in particular has no own identifier, the contained data is global.

Clusters Clusters are used as modules for grouping classes. In particular in Eiffel there are no visibility rules between clusters like in other programming languages. Clusters are modelled in the following way:

```
cluster(#id,'ClusterName').
```

- *#id* is the the primary key;
- *ClusterName* is the name of the cluster.

Class Declarations Classes are declared taking into account information about: its location given by the cluster it belongs to, its name and the list of its formal generics. The relation between class declaration facts and formal generic facts is one to many. If the class is not generic then it has no generic parameters and the list is empty. It is necessary to keep them in a list because their order is important for the consistency of the model. A simple reference from formal generic to parent class would not be sufficient.

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

- *#id* is the identifier of the class;
- *#cluster* is the identifier of the cluster the class belongs to;
- *ClassName* is the name of the class;
- *#formalGeneric* is the identifier of a formal generic parameter.

Obsolete Messages Obsolete messages are used for the issue of warnings in case the class they belong to, is deprecated.

```
obsoleteMessage(#classDecl or #routine,#manifestConstant)
```

- *#classDecl* is the identifier of the class;
- *#routine* is the identifier of the obsolete routine;
- *#manifestConstant* is the id of a string manifest constant containing the obsolete message of that class.

Index Clauses The index clauses are used to retain metadata about a class. A clause may have literal identifiers and the literal identifiers may have values. The values are expressed using manifest constant facts which will be modelled later on.

```
indexClause(#id,#classDecl).
```

- a class declaration may have zero or more index clauses;
- *#id* represents the primary key;
- #classDecl points to the class declaration to which the list belongs to.

The index clause may have an attached identifier. These facts are optional, so they are modelled as attribute facts for the index clause facts.

```
indexClauseIdentifierAttribute(#indexClause,'IdentifierName').
```

- an index clause may have an identifier attached;
- *IdentifierName* is the name of the identifier attached.

Index values facts model the values that are attached to index clauses:

```
indexValue(#id,#indexClause,#identifier or #manifestConstant).
```

- one index clause may have attached one or more index values;
- *#id* is the primary key;
- *#indexClause* the identifier of the clause referred;
- *#manifestConstant* is the identifier of the constant.

Class Declaration Attributes A class declaration may be augmented with several keywords deferred, expanded, separate, foster. These keywords are modelled through the following facts:

```
deferredClass(#classDecl).
expandedClass(#classDecl).
separateClass(#classDecl).
foster(#classDecl).
```

- all of these facts refer to class declarations;
- all clauses are optional for a class declaration.

2.1.2 Reification of Formal Generics

Formal generic parameters belong to a generic class. They are equipped with their unique identifier, next comes the identifier of the parent class and finally the name of the parameter.

```
formalGeneric(#id,#classDecl,'GenericParameterName').
```

- one classDecl may have zero or more formal generic parameters, it is correlated with the list from the foster class declaration;
- *#id* is the primary key;
- *#classDecl* represents the identifier of the class the generic parameter belongs to;
- *GenericParameterName* is the name of the generic parameter.

A formal generic may be restricted to a certain type, this constraint is modelled by the following fact:

```
constrainedClassType(#formalGeneric,#classType).
```

- a class type may have zero or more generic types;
- *#formalGeneric* represents the identifier of the generic parameter referred by the constraint class type;
- *#classType* is the identifier of the class type used for the constraint.

2.1.3 Reification of Inheritance

In this section we present the elements which reifies the ordinary inheritance class relationship. To express inheritance we need the identifier of the subclass and the identifier of the superclass type. It is a type and not just a simple class because in case of generic superclasses we have to provide also the actual generic parameters. This information belongs to the type and not to the instantiated class.

```
inheritance(#id,#classDecl,#classType).
```

- *#classDecl* represents the identifier of the current class which is the source of the inheritance relationship;
- *#classType* represents the identifier of the target class type of the relationship.

In inheritance the renaming of feature is possible, this aspect is modeled by the following Prolog fact:

```
rename(#id,#inheritance or #exheritance,#featureDecl,'FeatureNewName').
```

- *#id* is the key of the node;
- *#inheritance* is the identifier of the inheritance link the rename belongs to. Also we can notice that *#exheritance* is a valid choice, meaning that rename can be used in both inheritance and exheritance class relationships.
- *#featureDecl* is the identifier of the feature that is renamed;
- *FeatureNewName* is the new name of the feature.

In the context of inheritance, more specifically on the inheritance branches we can set the inherited features visibility by performing feature exportation. Between a feature and export client class there is a many to many relationship, meaning that a group of several features may be exported to a group of several classes.

```
exportInherit(#id,#inheritance).
```

- *#id* is the primary key;
- *#inheritance* is the inheritance branch identifier the export belongs to.

The next fact is used for modelling the classes that belong to a certain class group used for export purposes.

```
exportInheritClass(#id,#exportInherit,#classDecl).
```

- *#id* is the primary key;

- #exportExherit is the identifier of the export statement;
- *#classDecl* is the identifier of the class participating in the export statement.

Next we have the model of the features which are exported by the following fact:

```
exportInheritFeature(#id,#exportInherit,#featureDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;
- *#featureDecl* is the identifier of the feature that is exported.

The **all** keyword for an export statement is modelled by the following clause:

```
exportInheritFeatureAll(#id,#exportInherit).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement.

This means that all inherited features are exported to a certain group of classes.

When we want to undefine, redefine or select a feature on an inheritance branch we can use the following Prolog facts:

```
undefine(#id,#inheritance or #exheritance,#featureDecl).
redefineInherit(#id,#inheritance,#featureDecl).
selectInherit(#id,#inheritance,#featureDecl).
```

- *#id* is the key of the node;
- *#inheritance* or *#exheritance* is the identifier of the inheritance link that undefine, redefine or select belongs to;
- *#featureDecl* is the identifier of the feature referred from the superclass or exherited class.

2.1.4 Reification of Creators

The creators of an Eiffel class are modelled using the following fact:

```
creator(#id,#classDecl).
```

- *#id* is the primary key;
- *#classDecl* is the id of the class the creator belongs to.

A class has zero or more creators and that is why they have to be indexed.

The next fact models the features that are used for creation.

```
creatorFeature(#id,#creator,#procedure).
```

- *#id* is the primary key;
- *#creator* is the identifier of the creator the client the feature is accessible from;
- *#procedure* is the identifier of the feature that is declared as creator.

The client class that may use a creator can be set using the following fact:

```
creatorClientClass(#id,#creator,#classDecl).
```

- *#id* is the primary key;
- *#creator* is the identifier of the creator the client class can access;
- *#classDecl* represents the identifier of the client class that can access the creator feature.

2.1.5 Reification of Features

In this subsection we will present the facts which model the feature related entities. The feature block models a set of features grouped in the code using the **feature** keyword.

```
featureBlock(#id,#classDecl).
```

- *#id* is the primary key;
- *#classDecl* is the class the feature block belongs to.

The feature declaration fact models a feature declaration storing the identifier of the feature block and the feature name:

```
featureDecl(#id,#featureBlock,'FeatureName').
```

- *#id* is the primary key;
- *#featureBlock* refers to the feature block that the feature declaration belongs to.

The next fact models the characteristic of a feature to be an attribute:

```
attribute(#featureDecl).
```

- *#featureDecl* is the identifier of the feature.

The next two facts model the type of the operator when the feature denotes an operator by its name:

```
prefix(#featureDecl).
infix(#featureDecl).
```

The characteristic of a feature as being frozen is modelled using the following fact:

```
frozen(#featureDecl).
```

The following fact models the client classes for a block of features:

```
featureClientClass(#id,#featureBlock,#classDecl).
```

- *#id* is the primary key;
- *#featureBlock* is the identifier of the referred feature block;
- *#classDecl* is the identifier of the client class.

The formal argument list of a feature is modelled using the following fact:

```
formalArguments(#id,#featureDecl,[#formalArgument,...]).
```

- *#id* is the primary key;
- *#featureDecl* is the identifier of the referred feature owning the formal arguments;
- *#formalArgument* is the identifier of the formal argument from the ordered list.

The formal argument itself is modelled using the next fact, by storing the formal arguments fact which is the parent, the name and the type identifier:

```
formalArgument(#id,#formalArguments,'ArgumentName',#type).
```

- *#id* is the primary key;
- *#formalArguments* points to the feature the signature belongs to;
- *ArgumentName* is the name of the formal argument;
- *#type* is the identifier of the argument type.

The type mark of the feature represents its return type. Each feature may have attached one type mark, in case of procedures it is optional.

```
typeMark(#featureDecl,#type).
```

- *#featureDecl* points to the feature who owns the type mark;
- *#type* represents the return type of the feature.

Some features may have constant values attached to them in this case they are constant features in the class. To declare such a feature we will use the following definition:

```
featureManifestConstant(#featureDecl,#manifestConstant).
```

- *#featureDecl* is the identifier of the feature the constant belongs to;
- *#manifestConstant* is the identifier of the manifest constant attached to the feature.

To specify whether a feature is unique we can use the following rule:

```
unique(#featureDecl).
```

A routine can be attached to a feature by using the following fact:

```
routine(#id,#featureDecl).
```

- *#id* is the primary key;
- *#featureDecl* is the identifier of the feature the routine is attached to.
- The routine can be either deferred or once (meaning that it will be executed once in the runtime):

```
deferredFeature(#routine).
once(#routine).
```

- In Eiffel routines can be defined in other programming languages, so they need to have an external name in that language which can be different from the name of the feature in Eiffel. The external name of the feature is modelled as a String manifest constant.

```
external(#id,#routine,#manifestConstant).
```

- *#id* is the primary key;
- *#routine* is the routine that has the external implementation;
- *#manifestConstant* is the identifier of the String containing external name of the feature.
- The external alias of a feature can be modelled using the followings:

```
externalAlias(#external,#manifestConstant).
```

- #external is the id of the external declaration that is referred;

- #manifestConstant is the identifier of the String which is the name of the alias.

Local declarations in a routine may be declared using the following rule:

```
localDecl(#id,#routine,'LocalName',#type).
```

- *#id* is the primary key;
- *#routine* is the routine identifier the local declaration belongs to;
- *LocalName* the name of the local variable;
- *#type* points to the type identifier of the local variable.

There can be noticed that the order in which locals are declared is not important.

The next six clauses refer to all the possible assertions that can be added to a feature. The *require* fact is used for attaching the precondition of a feature within a class which has no superclass, except *ANY*. The *requireElse* is used for declaring the precondition of a feature in a class having one or more superclasses, completing the preconditions from the superclasses. We note that the precondition of the subclass must be weaker or equal than the precondition of the superclass. The *ensure* and *ensureThen* fact attaches a postcondition to a routine in the case of a class not having or having superclasses. The *requireOtherwise* and *ensureOtherwise* are used for declaring the precondition and postcondition of a feature in the foster class.

```
require(#id,#routine,#assertion).
requireElse(#id,#routine,#assertion).
requireOtherwise(#id,#routine,#assertion).
ensure(#id,#routine,#assertion).
ensureThen(#id,#routine,#assertion).
ensureOtherwise(#id,#routine,#assertion).
```

- *#id* is the primary key;
- *#routine* points to the routine the assertion belongs to;
- *#assertion* points to a fact modelling the logical expression to be verified.

The assertion fact is modelled as follows:

```
assertion(#id,#expression).
```

- #id is the primary key;
- *#expression* is the identifier of an expression which must return boolean value.

An assertion may have a tag as an attribute, this is modelled using the following fact:

```
assertionTagAttribute(#assertion,'TagName').
```

- *#assertion* is the identifier of the assertion;
- *TagName* is the name of the assertion tag.

The class type present in an adaptation item is modelled using the following clause:

```
adaptedClassType(#id,#attributeAdaptation or #methodAdaptation,#classType).
```

- *#id* is the primary key;

- *#attributeAdaptation* or *#methodAdaptation* are parent facts;
- *#classType* is the identifier of the class type used in the adaptation process.

The facts is used to adapt attributes or methods to a different signature or behavior.

The attribute adaptation is modelled using the following fact:

```
attributeAdaptation(#id,#featureDecl,#featureDecl,#type or
#identifier (like precursor),#expression,#expression).
```

- *#id* is the primary key;
- *#featureDecl* is the identifier of the parent feature which defines the adaptation;
- *#featureDecl* is the identifier of the feature from the subclass which is adapted;
- *#type* is the identifier of the result type;
- *#expression* is the identifier of the expression passed to precursor;
- *#expression* is the identifier of the expression passed representing the adapted result.

2.1.6 Reification of Types

The types are represented on two levels: at one level we have a fact modelling an abstract type and on the second level we have the concrete types:

```
type(#id,#classType or #expandedType or #separateType or #likeType or #bitType).
```

- *#id* is the primary key;
- the second argument (*#classType*, *#expandedType*, *#separateType*, *#likeType*, *#bitType*) is the identifier of the concrete type that the current fact refers to.

Class types are modelled by the following fact:

```
classType(#id,#classDecl or #formalGeneric).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the class name;
- *#formalGeneric* is the identifier of the formal generic parameter.

One can notice that the class type may refer a class declaration or a formal generic.

A class type may have actual generics if the class referred is generic. If the class type refers a formal generic it can never have actual generic types.

```
actualGenericType(#id,#classType,#formalGeneric,#type).
```

- *#id* is the primary key;
- *#classType* is the parent identifier the type the actual generic belongs to;
- *#formalGeneric* is the identifier of the formal generic parameter the actual generic type corresponds to;
- #type is the identifier of the actual generic type.

Expanded and separate types are a special kind of class types which have special properties. The expanded type instances are objects not references, while separate types are used in the concurent mechanisms of Eiffel.

```
expandedType(#id,#classType).
separateType(#id,#classType).
```

- *#id* is the primary key
- *#classType* is the identifier of the class type that is expanded/separate.

Like types or anchored types are types which refer to the types of other features, formal arguments or local declarations. There are allowed also links to **current** keyword which actually represents a contextual reference to the current object. These types were introduced mainly for helping covariant redeclaration of features:

```
likeType(#id,#identifier(current) or #featureDecl or #formalArgument or #localDecl).
```

- *#id* is the primary key;
- *#identifier* is the identifier of the current keyword;
- *#featureDecl* is the the identifier of the feature referred;
- *#formalArgument* is the identifier of the formal argument referred;
- *#localDecl* is the identifier of the local variable used.

Finally bit types are represented by the following fact. To be noted that they reference either a integer constant either a integer constant feature.

```
bitType(#id,#integerManifestConstant or #featureDecl).
```

- *#id* is the primary key;
- *#integerManifestConstant* refers to an integer constant;
- *#featureDecl* refers to an integer constant attribute.

2.1.7 Reification of Instructions

A compound instruction is modelled as a ordered set of instructions by the following fact:

```
compound(#id,
 #routine or #conditional or #elseIf or #conditionalElse or
 #when or #multiBranchElse or #loop or #debug,
 [#creation,#assign,#reverse,#call,#conditional,#multiBranch,#loop,
  #debug,#check,#retry]).
```

- *#id* is the primary key;
- the parent entity may be *#routine*, *#conditional*, *#elseIf*, *#when*, *#multiBranchElse*, *#loop*, *#debug*;
- refers in order instruction clauses like *#creation*, *#call*, *#multiBranch*, *#loop*, *#debug*, *#check*, *#retry.*

A rescue compound is modelled like this:

```
rescueCompound(#id,#routine,
 [#creation,#assign,#reverse,#call,#conditional,#multiBranch,#loop,
  #debug,#check,#retry]).
```

- *#id* is the primary key;
- the parent entity may be routine only;
- refers in order instruction clauses like *#creation*, *#call*, *#multiBranch*, *#loop*, *#debug*, *#check*, *#retry*.

The creation instruction is modelled next:

```
creation(#id,#compound,#featureDecl or #localDecl or #identifier (result)).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the creation instruction belongs to;
- *#featureDecl* is the identifier of the feature referencing the newly created object;
- *#localDecl* is the identifier of the local referencing the newly created object;
- *#identifier* is the identifier of the result keyword referencing the newly created object.

The type of the creation instruction can be expressed optionally using the next fact:

```
creationType(#id,#creation,#type).
```

- *#id* is the primary key;
- *#creation* is the creation instruction the type is used for;
- *#type* is the type used in the creation.

The creation instruction may imply also a call which is modeled next:

```
creationCall(#id,#creation,#call).
```

- *#id* is the primary key;
- *#creation* is the creation instruction the call is used in;
- *#call* is the identifier of the call needed in the creation process.

The assignment instructions are represented by the following facts:

```
assign(#id,#compound,#featureDecl or #localDecl or #identifier (result),
#expression).
reverse(#id,#compound,#featureDecl or #localDecl or #identifier (result),
#expression).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the assignment belongs to;
- *#featureDecl* is the identifier of the assigned feature, which must denote an attribute;
- *#localDecl* is the identifier of the assigned local;

- *#identifier* is the identifier of the result keyword;
- *#expression* is the identifier of the expression assigned.

A conditional instruction has a parent, an expression that must be evaluated and a compound. Optionally there might be present some ordered else-if instructions which deal with other alternatives:

```
conditional(#id,#compound,#expression,#compound,[#elseif,...]).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the conditional belongs to;
- *#expression* is the identifier of the evaluated expression;
- *#compound* is the identifier of the compound instruction executed on the then branch;
- *#elseif* is the identifier of the ordered else-if instructions.

The else-if instruction contains a condition, an expression and a compound.

```
elseIf(#id,#conditional,#expression,#compound).
```

- *#id* is the primary key;
- *#conditional* is the instruction the elseif belongs to;
- *#expression* is the identifier of the evaluated expression;
- *#compound* is the identifier of the compound instruction executed on the else-if branch.

The conditional else branch of a conditional instruction is modelled next:

```
conditionalElse(#id,#conditional,#compound).
```

- *#id* is the primary key;
- *#conditional* is the instruction the else belongs to;
- *#compound* is the identifier of the compound instruction executed on the else branch.

The multi branch instruction is similar to the switch instruction of C language and it is modelled as follows:

```
multiBranch(#id,#compound,#expression,[#when,...]).
```

- *#id* is the primary key;
- *# compound* is the identifier of the parent the creation belongs to;
- *#expression* is the identifier of the evaluated expression;
- *#when* is the identifier of the when clause in the ordered list.

The branches are modelled by the following clause:

```
when(#id,#multiBranch,#compound).
```

- *#id* is the primary key;
- *#multiBranch* is the identifier of the branch the when clause belongs to;

- *#compound* is the identifier of the compound instruction to be executed if one of the when choices match.

The choice element is modelled next:

```
choice(#id,#when).
```

- *#id* is the primary key;
- *#when* is the identifier of the parent branch.

A choice may be represented either by constants either by constant ranges:

```
choiceConstant(#id,#choice,#featureDecl or #manifestConstant).
choiceConstantRange(#id,#choice,#featureDecl or #manifestConstant,
 #featureDecl or #manifestConstant).
```

- *#id* is the primary key;
- *#choice* is the identifier of the parent choice;
- third and fourth arguments are identifiers of constant attributes or manifest constants (character or integer).

The multi branch instruction is equipped with an optional else alternative, as we can see next:

```
multiBranchElse(#id,#multiBranch,#compound).
```

- *#id* is the primary key;
- *#multiBranch* is the identifier of the parent multiple branch instruction;
- *#compound* is the identifier of the compound instruction to be executed in case no branch is taken.

The only instruction of Eiffel dealing with iteration is modelled next:

```
loop(#id,#compound,#compound,#expression,#compound).
```

- *#id* is the primary key;
- second argument *#compound* the is the identifier of the parent the creation belongs to;
- third argument *#compound* is the identifier of the compound instruction to be executed for initialization purposes;
- *#expression* is the identifier of the expression to be evaluated at each step;
- #compound is the identifier of the compound instruction to be executed at each step.

The invariant of a looping instruction follows:

```
loopInvariant(#id,#loop,#assertion).
```

- *#id* is the primary key;
- *#loop* is the identifier of the parent loop instruction;
- *#assertion* is the identifier of the assertion representing the loop invariant.

A loop instruction may have variant expressions also:

```
loopVariant(#id,#loop,#expression).
```

- *#id* is the primary key;
- *#loop* is the identifier of the parent loop instruction;
- *#expression* is the identifier of the expression representing the loop variant.

The variant may have an name attached optionally:

```
loopVariantIdentifierAttribute(#loopVariant,'VariantName').
```

- *#loop* is the id of the parent loop variant;
- '*VariantName*' is the name of the variant.

The debug instruction is the next one modelled:

```
debug(#id,#compound,#compound).
```

- *#id* is the primary key;
- the second argument *#compound* is the identifier of the parent the creation belongs to;
- the third argument *#compound* is the id of the compound instruction executed in case of debug situations.

A debug instruction may have attached zero or more keys:

```
debugKey(#id,#debug,#manifestConstant).
```

- *#debug* is the identifier of the parent debug instruction;
- *#manifestConstant* is the identifier of the string value of the debug key.

The check instruction verifies an assertion related to a compound instruction:

```
check(#id,#compound,[#checkAssertion,...]).
```

- *#id* is the primary key;
- *#compound* is the identifier of the parent the creation belongs to;
- *#checkAssertion* refers to an assertion that will be checked at runtime. They are kept in a list in order to preserve their execution order.

The assertion of a check instruction is handled by the following fact:

```
checkAssertion(#id,#check,#assertion).
```

- *#id* is the primary key;
- *#check* is the identifier of the parent check instruction fact;
- *#assertion* is the identifier of the assertion to be checked at runtime.

The retry instruction is used in case of a constraint failure in a compound. As the retry instruction has no parameters it can be unique in the model and it can be referred as many times as necessary.

```
retry(#id,#compound).
```

- *#id* is the primary key;

- *#compound* is the identifier of the parent the creation belongs to.

Calls are a special kind of instructions. They are modelled in a quite sophisticated way because of the composed receivers:

```
call(#id,#expression or #compound, #featureDecl or #localDecl).
```

- *#id* is the primary key;
- the second argument *#expression* or *#compound* is the identifier of the parent the creation belongs to;
- *#featureDecl* is the identifier of the feature declaration that is called, in case of precursor call, it is the id of the feature from the superclass or an identifier representing a local.

The call receivers represent the constructions before the called feature and they are linked one to another in case there are multiple ones:

```
callReceiver(#id,#call or #callReceiver,
 #identifier (result, current, precursor) or
 #expression or #call or #featureDecl or #formalArgument or #localDecl).
```

- #id is the primary key;
- the second argument models the parent of the receiver:
  - *#call* is the identifier of the call the receiver belongs to;
  - *#callReceiver* is the identifier of the parent call receiver of the current receiver;
- the third argument models the receiver entity, and it can be:
  - *#identifier* points to **result**, **current** or **precursor** keywords;
  - these identifiers will be always the last call receivers;
- *#expression* points to an expression identifier;
- *#call* is the identifier of another parent call;
- *#featureDecl* is the identifier of a feature referencing an object;
- *#formalArgument* is the identifier of a formal argument referencing an object;
- *#localDecl* is the identifier of a local declaration referencing an object.

For example the call *current.f.g.h* will be represented like this:

```
call(200,100,id of feature h).
callReceiver(300,200,id of feature g).
callReceiver(301,300,id of feature f).
callReceiver(302,301,id of "current").
```

The precursor receiver of a call may be parameterized by a certain type. This is useful in case of multiple inheritance to select a certain implementation to be executed from one parent.

```
callReceiverPrecursorTypeAttribute(#id,#callReceiver,#type).
```

- *#id* is the primary key;
- *#callReceiver* is the parent receiver which can be only **precursor**;

- *#type* is the identifier of the type used for casting.

The actuals of a call are modelled next. They are not ordered since they refer the formal argument of the feature where the order is kept:

```
actual(#id,
 #creationCall or #call or #actuals,
 #formalArgument,
 #expression or #featureName or #identifier (current, result)).
```

- *#id* is the primary key;
- *#creationCall* or *#call* identifies the call using the current call actual;
- *#actuals* identifies a fact modelling a set of actuals;
- *#formalArgument* the identifier of the corresponding formal argument;
- *#expression* is the identifier of the expression used as actual parameter;
- *#featureName* is the identifier of the feature representing the address mark;
- *#identifier* may be identifier of **current** or **result** representing address marks.

2.1.8 Reification of Expressions

Expressions are modelled through a fact which may pointing at several facts representing expressions:

```
expression(#id,#call or #identifier (current, result, precursor) or
 #subexpression or #manifestConstant or #manifestArray or
 #unaryOperator or #binaryOperator or #strip).
```

- *#id* is the primary key;
- the second argument is the identifier of another subexpression element:
  - *#call* is the id of a call;
  - *#identifier* which may point to **current** or **result** keywords;
  - *#subexpression* points to another expression;
  - *#manifestConstant* points to a constant;
  - *#manifestArray* is the identifier of an array operator;
  - *#unaryOperator* is the identifier of an unary operator;
  - *#binaryOperator* is the identifier of a binary operator;
  - *#strip* is the identifier of a strip expression.

Subexpressions model an expression which is enclosed between parenthesis:

```
subexpression(#id,#expression).
```

- *#id* is the primary key;
- the second argument is the identifier of the expression element.

As component of an expression is the unary operator. It may be: +, -, not, free operator, old.

```
unaryOperator(#id,'OperatorSymbol',#expression).
```

- *#id* is the primary key;
- *'OperatorSymbol'* is the symbol of the operator;
- *#expression* is the identifier of the expression representing the operators argument.

Another expression component is the binary operator (free operator symbol, +, -, *, /, ^, div, mod, =, /=, <, >, <=, >=, and, or, xor, and then, or else, implies):

```
binaryOperator(#id,#expression,'OperatorSymbol',#expression).
```

- *#id* is the primary key;
- #expression is the identifier of the expression representing the operators left argument;
- *'OperatorSymbol'* is the symbol of the operator;
- *#expression* is the identifier of the expression representing the operators right argument.

The manifest array expression is modelled by the next fact:

```
manifestArray(#id,[#manifestArrayExpression,...]).
```

- *#id* is the primary key;
- the third argument is a table for maintaining the order of the expression identifiers.

The manifest array expression is modelled by the next fact:

```
manifestArrayExpression(#id,#manifestArray,#expression).
```

- *#id* is the primary key;
- *#arrayOperator* is the identifier of the array operator referred;
- *#expression* is the identifier of the expression used by the operator.

A strip expression returns an array of all attributes of an object. It is modelled by the following fact:

```
strip(#id).
```

- *#id* is the primary key of the strip expression.

A strip attribute is modelled next:

```
stripAttribute(#id,#strip,#featureDecl).
```

- *#id* is the primary key;
- *#strip* is the parent strip instruction the attribute belongs to;
- *#featureDecl* is the identifier of the referred attribute.

Identifiers are modelled by the following facts:

```
identifier(#id,'IdentifierName').
```

- *#id* is the primary key;
- *'IdentifierName'* is the name of the identifier.

Manifest constants of type boolean, character, integer, real, string, bit are modelled next:

```
manifestConstant(#id,'ManifestConstantValue', 'boolean' or 'character' or
'integer' or 'real' or 'string' or 'bit').
```

- *#id* is the primary key;
- *'ConstantValue'* is the value of the manifest constant;
- the third argument models the type of the constant.

Invariants are boolean expressions which must hold in the context of a class:

```
invariant(#id,#classDecl,#assertion).
```

- *#id* is the key of the node;
- *#classDecl* is the identifier of the class that the invariant refers to;
- *#assertion* is the identifier of the invariant assertion.

2.2 Reification of Reverse Inheritance

In this section we will present the foster class grammar emphasizing only the aspects which are newly added to the ordinary Eiffel grammar. These syntactical elements were already discussed in the context of semantics definition [Chi08], and next they are modelled for the selected implementation.

In contrast with ordinary classes, only the text of foster classes is affected by reverse inheritance elements. All the reverse inheritance declarations, which are used to build a certain class hierarchy, are written only in the text of foster classes. In the foster class we can find all the old and new mechanisms of the reverse inheritance semantics. The old mechanisms are taken from ordinary inheritance and changing their semantics like: undefine, redefine, export, select, assertions and also genuine mechanisms like factorization, adaptation.

We can admit that the model of reverse inheritance class relationship is equivalent to the model of the foster class. The model of the foster class includes all elements from the model of the ordinary Eiffel class and the reverse inheritance extension elements as it can be noticed in appendix A.

Since in ordinary inheritance the text of a class is written upon certain syntactical and semantical rules checked by the Eiffel compiler, in reverse inheritance the foster class syntax is checked by our parser but the semantical checks are performed at the factbase level.

2.2.1 Reification of Exheritance

The exheritance class relationship is expressed between the foster class and the exherited classes as types and instantiated in case of generic classes:

```
exheritance(#id,#classDecl,#classType).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the current, source class;
- *#classType* is the identifier of the exherited class type.

The adapt and moveup clauses are linked to the exheritance class relationship and to a feature:

```
adapt(#id,#exheritance,#featureDecl).
moveup(#id,#exheritance,#featureDecl).
```

- *#id* is the primary key;
- *#exheritance* is the identifier of the exheritance branch;
- *#featureDecl* is the identifier of the feature to be adapted.

The exheritance selection mechanism is presented next:

```
selectExherit(#id,#exheritance,#descendantQualification).
```

- *#id* is the primary key;
- *#exheritance* is the identifier of the exheritance branch;
- *#descendentQualification* is the identifier of the class chain that the feature is selected in.

The feature which may be selected in the exheritance selection mechanism is next. One exheritance clause may have multiple features to be selected.

```
selectExheritFeature(#id,#selectExherit,#featureDecl).
```

- *#id* is the primary key;
- *#selectExherit* is the identifier of the parent fact;
- *#featureDecl* is the identifier of the feature to be selected.

Descendant class chains can be constructed using the following clause:

```
descendantQualification(#id,#descendantQualification,#classDecl).
```

- *#id* is the primary key;
- *#descendantQualification* is the element before the current one, the first entity will have no predecessor, so this value will be zero;
- *#classDecl* is the identifier of the class in the chain.

The export declarations in the context of reverse inheritance are attached to the class and not to the inheritance clause, this is why they are modelled separately. Between a feature and export client class there is a many to many relationship (multiple features can be exported to multiple classes):

```
exportExherit(#id,#classDecl).
```

- models an export statement;
- *#id* is the primary key;
- *#classDecl* is the foster class identifier the export belongs to.

The classes which may be linked to an export statement are modelled next:

```
exportExheritClass(#id,#exportExherit,#classDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;

- *#classDecl* is the identifier of the class participating in the export statement.

Exported features are attached to the export statement by the following fact:

```
exportExheritFeature(#id,#exportExherit,#featureDecl).
```

- *#id* is the primary key;
- *#exportExherit* is the identifier of the export statement;
- *#featureDecl* is the identifier of the feature that is exported.

If one desires to export all features of a class the following clause must be attached to the export clause:

```
exportExheritFeatureAll(#id,#exportExherit).
```

- #id is the primary key;
- #exportExherit is the identifier of the export statement.

The redefinition in the context of reverse inheritance is global, it does not belong to an exheritance branch like in ordinary inheritance:

```
redefineExherit(#id,#classDecl,#featureDecl).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the class containing the feature;
- *#featureDecl* is the identifier of the redefined feature.

The selection mechanism of exheritance allows to select or deny a set of specific features through the following facts:

```
onlyFeature(#id,#classDecl,#featureDecl).
exceptFeature(#id,#classDecl,#featureDecl).
```

- *#id* is the primary key;
- *#classDecl* is the identifier of the class hosting the feature selection clauses;
- *#featureDecl* is the feature in the exherited class.

The selection mechanism can be set to select all exheritable features or no features at all, in order to create a new type:

```
allFeature(#classDecl).
nothingFeature(#classDecl).
```

- *#classDecl* is the identifier of the class hosting the feature selection clause.

2.2.2 Reification of Routine Adaptation

A rather new concept, but very necessary in the process of feature exheritance is the concept of adaptation, which changes dramatically the structure of an ordinary feature. The main change consists, as we know, in having a feature adaptation clause for each exherited class when necessary.

```
routineAdaptation(#id,#featureDecl,#featureDecl).
```

- *#id* is the primary key
- *#featureDecl* is the identifier of the feature which defines the adaptation;
- *#featureDecl* is the identifier of the feature from the subclass which is adapted.

The adapted formals are modelled next:

```
adaptedFormals(#routineAdaptation,#formalArguments or #identifier (like precursor)).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#formalArguments* is the identifier of the formal arguments fact;
- *#identifier* is the identifier of **like precursor** keyword construction.

The adapted type mark is modelled next:

```
adaptedTypeMark(#routineAdaptation,#type or #identifier (like precursor)).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#type* is the identifier of the return type of the method;
- *#identifier* is the identifier of the **like precursor** keyword construction;

The next fact models the adapted actuals of an adapted feature, they can be regular actuals or the **precursor** keyword.

```
adaptedActuals(#routineAdaptation,#actuals or #identifer (precursor)).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#actuals* is the identifier of the set of actuals;
- *#identifier* (**precursor**) is the identifier of the keyword used when a simple call has to be made.

The fact refers to the adapted actuals:

```
actuals(#id,#adaptedActuals).
```

- *#id* is the global identifier for the actuals;
- *#adaptedActuals* is the parent fact.

The adapted result consists in an expression which may contain the **result** keyword and is modelled by the following fact:

```
adaptedResult(#routineAdaptation,#expression).
```

- *#routineAdaptation* is the identifier of the parent fact;
- *#expression* is the identifier of the expression fact.

Example 1 RIEiffel Type Rules

```
%classDecl(#id,#cluster,'ClassName',[#formalGeneric,...]).
ast_node_def('RIEiffel',classDecl,[
 ast_arg(id,             mult(1,1,no),  id,   [classDecl]),
 ast_arg(parent,         mult(1,1,no),  id,   [cluster]),
 ast_arg(className,      mult(1,1,no),  attr, [atom]),
 ast_arg(formalGenerics, mult(1,*,ord), id,   [formalGeneric])
]).
```

2.3 Metamodel Validity Rules

It is very natural to validate the reverse inheritance metamodel by a set of rules in order to be able to check its consistency before the transformation. We intend to define only the validity rules regarding the reverse inheritance related mechanisms: feature selection, redefinition, adaptation, selection, assertion. From the technical point of view, validity rules are expressed as Prolog predicates operating on the model factbase. For some rules we present their code in Prolog.

2.3.1 Type Checking

Any RIEiffel factbase is checked against a formal metamodel where all facts are described together with their arguments. Each argument has a type information allowing type checking. For example, the class declaration fact has as fourth argument a list of formal generic identifiers. The type checker will verify that each fact corresponding to that set of identifiers is a formal generic and not some other fact.

In example 1 we show the node definition for a class declaration. The definition contains information about the language name (in this case *RIEiffel*), node name (*classDecl*) and each fact parameter. The first argument is the unique identifier of the fact, named *id*, its multiplicity is one to one with the fact, not ordered, its kind is identifier and its type is *classDecl*. The second argument is named *parent*, the current fact has a one to one multiplicity relation with the cluster fact, its kind is identifier, and the type is *cluster*. The third argument is named className, has a one to one multiplicity relation with the attribute, its kind is attribute and the type is *atom*. The fourth argument is named formalGenerics, the current fact has a one to many multiplicity with the formal generic facts, it is an ordered set (*ord*), its kind is identifier and the type is *formalGeneric*.

2.3.2 Exherited Features Validity Rules

Regarding the aspect of feature factorization, several validity rules must hold on the factbase model.

2.3.2.1 Single Selection Rule

The first rule from example 2 verifies that only one exherited feature selection mechanism of **all**, **only**, **except**, **nothing** will be used in a foster class.

2.3.2.2 Exheritable Selection Rule

All explicitly selected features using **only**, **except** keywords from the exherited classes must exists and must have compatible signatures in all the exherited classes. In other words it does not make sense explicitly selecting or unselecting sets of features which are not exheritable.

Example 2 Single Selection Rule

```
checkSingleSelectionMechanism(FosterClassId):-
 exists(allFeature(FosterClassId))->
  not(exists(onlyFeature(_,FosterClassId,_));
  exists(exceptFeature(_,FosterClassId,_));
  exists(nothingFeature(FosterClassId))),
 !.
checkSingleSelectionMechanism(FosterClassId):-
 exists(onlyFeature(_,FosterClassId,_))->
  not(exists(allFeature(FosterClassId));
  exists(exceptFeature(_,FosterClassId,_));
  exists(nothingFeature(FosterClassId))),
  !.
checkSingleSelectionMechanism(FosterClassId):-
 exists(exceptFeature(_,FosterClassId,_))->
  not(exists(allFeature(FosterClassId));
  exists(onlyFeature(_,FosterClassId,_));
  exists(nothingFeature(FosterClassId))),
  !.
checkSingleSelectionMechanism(FosterClassId):-
 exists(nothingFeature(FosterClassId))->
  not(exists(allFeature(FosterClassId));
  exists(exceptFeature(_,FosterClassId,_));
  exists(onlyFeature(_,FosterClassId,_))),
  !.
```

2.3.2.3 Non-conflicting Selection Rule

The selected features must not generate conflicts with the new features of the foster class. This means that we can not select a feature for exheritance while it already exists in the foster class. In such a case the foster class should be designed with a redefinition clause.

2.3.2.4 Immediate Feature Selection Rule

The exherited features must exist locally in the subclasses and not inherited by some ancestor which is not the parent of the foster class. Thus, we avoid creating new links between already existing classes.

2.3.3 Exherited Feature Redefinition Validity Rules

In the case of feature redefinition, performed from the foster class, several rules must be applied:

- for each redefined set of features from the exherited classes, a new feature must exist in the foster class. This means that the set of *redefineExherit* clauses must point to features present in all exherited classes, having the same final name relative to the foster class.
- the redefined set of features from the exherited classes must have covariant signatures with the newly defined feature from the foster class;
- a redefined attribute in the foster class can not have corresponding method candidate features in the subclasses;
- a redefined attribute from the foster class can not have corresponding deferred candidate features in the subclasses.

2.3.4 Exherited Feature Adaptation Validity Rules

Adapted features obey to the following rules:

- an adapted feature must have formal arguments or return type, otherwise there is nothing to adapt;
- one feature listed in the adaptation clause must have at least one conversion sequence in the foster class corresponding to an exherited class;
- the signature of the conversion sequence must be identical with the one from the subclass;
- the new implementation signature must be equal to the one of the new feature from the foster class.

2.3.5 Exherited Feature Implementation Migration Validity Rules

The **moveup** mechanism is driven by the following rules:

- in a foster class the **moveup** clause for an exherited feature set can be used only on one exheritance branch. It does not make sense selecting multiple implementations for the same feature to migrate in the foster class thus arising a conflict.
- the dependencies of the moved features must be provided in the foster class by exheritance or by redefinition;
- for one exherited feature it can not be applied both moveup and adapt;
- a feature can not be both moved up and redefined.

2.3.6 Exherited Feature Selection Validity Rules

In the case of repeated inheritance the selection of replicated features must satisfy the next rules:

- the selected feature must belong to one of the exherited feature sets;
- all the classes from the selection clause must be direct or indirect subclasses of the foster class enclosing the select declaration.

2.3.7 Formal Generics Validity Rules

Since the foster class and the exherited classes may or not be generic and genericity can be constrained and unconstrained, we will set rules for all possible combinations.

2.3.7.1 Non-generic Foster Class and Generic Subclasses

When the foster class is non-generic, it will refer to the exherited classes through class types which may have attached actual generics. Considering this, we issue the following rules:

- all formal generics of exherited classes must be instantiated with the question mark symbol "**?**" and not with a class type. Since these actual generics will be lost in the equivalent class hierarchy based on ordinary inheritance, we decided in [Chi08] to use a special syntax instantiating the subclasses with the question mark symbol **"?**".
- not all subclasses must be generic.

2.3.7.2 Generic Foster Class and Generic Subclasses

In this case the instantiation information for the exherited classes will be reused for instantiating the foster class. Thus the following rules apply:

- each formal argument of the foster class must instantiate a formal generic in all exherited classes;
- if the formal generics of the foster class are constrained then the instantiated ones from the exherited classes must have the same constraint;

2.3.7.3 Generic Foster Class and Non-generic Subclasses

When the foster class is generic, unconstrained or constrained, the subclasses are non-generic and there is no instantiation information available, this is considered an invalid case, like discussed in [Chi08].

2.4 Summary

In this chapter we presented the metamodel of the RIEiffel language. For each semnificative grammar rule, a Prolog fact to represent a certain language entity is designed. We have to admit that the metamodel design is not homogeneous because some facts have relations with their parents in both ways and other facts only in one direction. This decision was taken in order to optimize the model, to minimize the number of facts and to capture some semantical aspects. The link between *formalArguments* and *formalArgument* is bidirectional and also the children are ordered in the parent, since argument order may not be ignored. The language entities were modeled as flexible facts linked together by identifiers. Adding and removing language entities are expressed at model level by adding or removing facts.

The factbase model is structured in two main parts: Eiffel language related facts and RIEiffel specific facts. The Eiffel facts are structured as follows: class header, formal generics, inheritance clauses, creators, features, instructions, expressions. The RIEiffel model contains the exheritance clauses facts and the adaptation facts. Still some details like spaces and comments are not modelled in our design since they have no semantical value. As a consequence the reverse engineered sources may not have the same text organization.

A Prolog metamodel which consists in describing each rule arguments, allows checking automatically the type consistency of the factbase. In the metamodel are also included navigation information for the nodes to be displayed in a graphic user interface. Validity rules are stated around the reverse inheritance concepts like: feature factorization, redefinition, adaptation, selection, exportation and around some complex contexts of reverse and ordinary inheritance. These rules substitute the semantics analysis which should be performed by a compiler against the reverse inheritance extension of a class.

Chapter 3

Model Transformations

In this chapter we present the model transformations that must be performed on the factbase model representing Eiffel class hierarchies built with reverse inheritance in order to generate an equivalent model by eliminating the new class relationship. From the use cases of reverse inheritance described in [Chi06, Chi08], we identify two main class configurations:

1. class hierarchies in which the foster class has no superclass, except class *ANY* which may be ignored if the foster class does not redefine features from it;
2. class hierarchies in which the foster class is "in the middle", it has both superclass(es)[1] and subclasses.

We start developing the transformations for the first class configuration, which we consider that they are basic transformations and then we will adapt them to fit to the second class configuration. Each factbase model transformation is presented systematically by providing an example and commenting the transformation code. At the end of this chapter in a special section we present how all transformations are composed and how they work assembled altogether.

3.1 Conditional Transformations

Conditional transformations (CTs) are defined and explained in [KK02, Kni06]. A CT is an abstraction composed of a condition and a transformation under a well defined set of formal rules. The CT based solution we are using in our program transformation is relying on:

- the representation of programs and models as logic factbases, as it was described in chapter 2;
- condition evaluation based on the declarative semantics of logic programs;
- information propagation via sets of substitutions computed for logic variables;
- the transformation of logic factbases.

Such an approach is applicable to arbitrary languages, models and artifacts. In example 3 is presented the structure of a CT. Technically, it is defined as a Prolog fact called **ct**. It has three terms: the **name** term which is arbitrary, the **condition** term and the **transformation** term. The name may have attached a list of arguments representing the parameters of the CT. The **condition** term will have a sequence of facts representing the conditions, while the **transformation** term will have a sequence of predicates for creation and deletion of facts from the factbase. The predefined predicates for the factbase manipulation are named **add** and **delete**.

[1] The superclass(es) will not add new features to the foster class in order to keep unmodified the behavior of the exherited classes. This requirement is necessary since reverse inheritance was designed as a non-destructive class relationship [Chi08].

Example 3 Conditional Transformation Structure

```
ct(
 name(arg1,arg2,...),
 condition(cond1,cond2,...),
 transformation(transf1,transf2,...)
).
```

Example 4 Conditional Transformation Example

```
ct(
 removeFoster(FosterClassId),
 condition((
  exists(classDecl(FosterClassId,_,_,_)),
  exists(foster(FosterClassId))
 )),
 transformation(
  delete(foster(FosterClassId))
 )
).
```

For example, the location of foster classes and the deletion of their foster attribute may be viewed in example 4. The name of the CT is *removeFoster*, the condition sequence calls two *exists* facts and the transformation uses the predefined **delete** predicate to erase facts from the factbase. The transformations will be applied to all sets of values selected by the condition sequence.

The parameters of a CT can be of several types: input, output and input-output, like in any Prolog rule. The CTs communicate one with another through parameters and the propagation of information is determined by the operator. CTs are linked together in a sequence through special binary operators **ANDSEQ**, **ORSEQ** and **PROPSEQ**. The operators have two operands, right-hand side and left-hand side, which may be either CT or another operator.

3.1.1 ANDSEQ Operator

The ANDSEQ operator propagates the results forward from the first CT to the second one and also backward if necessary. The parameter value sets for which the condition fails in the second CT will be back-propagated to the first CT undoing its transformations. For example, a situation where such an operator is necessary is when we create a feature block within a foster class using a first CT while the second CT is responsible with the creation of the exherited features. If there is no feature to exherit then the creation of the feature block is cancelled due to the back-propagation nature of the ANDSEQ operator. As a consequence such an operator requires that the first operand must have output parameters which are connected to the input parameters of the second operand.

3.1.2 PROPSEQ Operator

The PROPSEQ operator propagates forward the values of the parameters from the first operand to the second one, but never cancels any transformations executed by the first one. For example, such an operator must be used when we compute the exherited features and only for them we want to create formal arguments.

Example 5 Feature Exheritance (Original Eiffel Code)

```
foster class FC
exherit
 EC1
 EC2
 EC3
 all
end
class EC1
feature
 f(a:T1;b:T2;c:T3):T1 is do end
end
class EC2
feature
 f(x:T1;y:T2;z:T3):T1 is do end
end
class EC3
feature
 f(m:T1;n:T2;p:T3):T1 is do end
end
class T1 end
class T2 end
class T3 end
```

3.1.3 ORSEQ Operator

The ORSEQ operator does not propagate anything from left-hand side operand to right-hand side operand like the previous two ones did. This operator just takes any propagated input from the context and feeds both inputs of the referred CTs. Such an operator can be used when the two CTs do not interact one with the another. For example the deletion of exheritance clauses and the deletion of the foster class status are independent and they do not need to interact.

3.2 Feature Exheritance

In this section we will discuss technical issues regarding feature exheritance. Feature exheritance may be considered the core of the reverse inheritance semantics and all transformations are built around this mechanism. The exheritance mechanism is based on the fact that exherited features have the same or compatible signatures. This means that they must have the same name signatures in all exherited classes. In case of redefinition the signatures from the subclasses must conform to the new signature written in the foster class. When signatures are compared, the final features names are considered, after the effect of possible renamings. The set of exheritable features may be reduced by the **only** and **except** mechanisms.

We start with a simple exheritance case where the exherited features have both formal arguments and return types and all the types from the signature are class types.

We present the original context of the class hierarchy in Eiffel code in example 5. There are involved three exherited classes *EC1*, *EC2*, *EC3* and in each class there is one feature *f*, which is perfectly exheritable because they have the same signature. The method signature has three formal arguments and a return type. The foster class *FC* selects for exheritance any available features.

In example 6 we present the equivalent Prolog clauses which model the code from example 5.

In example 7 we can notice that there were added new facts which define the structure of the transformed foster class. Feature f is a new feature in the foster class *FC* and it is declared as

Example 6 Feature Exheritance (Original Prolog Clauses)

```
exists(classDecl(100,1,'FC',[])).  exists(classDecl(101,1,'EC1',[])).
exists(classDecl(102,1,'EC2',[])). exists(classDecl(103,1,'EC3',[])).
exists(foster(100)).               exists(exheritance(501,100,111)).
exists(exheritance(502,100,112)).  exists(exheritance(503,100,113)).
exists(allFeature(100)).           exists(featureBlock(701,101)).
exists(featureBlock(702,102)).     exists(featureBlock(703,103)).
exists(featureDecl(711,701,'f')).  exists(featureDecl(712,702,'f')).
exists(featureDecl(713,703,'f')).
exists(formalArguments(851,711,[861,862,863])).
exists(formalArguments(852,712,[871,872,873])).
exists(formalArguments(853,713,[881,882,883])).
exists(formalArgument(861,851,'a',1101)).exists(formalArgument(862,851,'b',1102)).
exists(formalArgument(863,851,'c',1103)).exists(formalArgument(871,852,'x',1101)).
exists(formalArgument(872,852,'y',1102)).exists(formalArgument(873,852,'z',1103)).
exists(formalArgument(881,853,'m',1101)).exists(formalArgument(882,853,'n',1102)).
exists(formalArgument(883,853,'p',1103)).
exists(typeMark(711,1101)).   exists(typeMark(712,1101)).
exists(typeMark(713,1101)).   exists(routine(751,711)).
exists(routine(752,712)).     exists(routine(753,713)).
exists(compound(801,751,[])). exists(compound(802,752,[])).
exists(compound(803,753,[])). exists(type(1101,1201)).
exists(type(1102,1202)).      exists(type(1103,1203)).
exists(classType(111,101)).   exists(classType(112,102)).
exists(classType(113,103)).   exists(classType(1201,1301)).
exists(classType(1202,1302)). exists(classType(1203,1303)).
exists(classDecl(1301,1,'T1',[])).  exists(classDecl(1302,1,'T2',[])).
exists(classDecl(1303,1,'T3',[])).
```

Example 7 Feature Exheritance (Transformed Prolog Clauses)

```
% new facts
exists(featureBlock(100008, 100)). exists(featureDecl(100009, 100008, f)).
exists(routine(100010, 100009)).   exists(deferredFeature(100010)).
exists(formalArguments(100021, 100009, [100015, 100016, 100017])).
exists(formalArgument(100015, 100021, a, 1101)).
exists(formalArgument(100016, 100021, b, 1102)).
exists(formalArgument(100017, 100021, c, 1103)).
exists(typeMark(100009, 1101)).
```

Example 8 Feature Exheritance (Transformed Eiffel Code)

```
deferred class FC
feature
 f(a:T1;b:T2;c:T3):T1
 is deferred end
end
class EC1
inherit
 FC
feature
 f(a:T1;b:T2;c:T3):T1
 is do end
end
class EC2
inherit
 FC
feature
 f(x:T1;y:T2;z:T3):T1
 is do end
end
class EC3
inherit
 FC
feature
 f(m:T1;n:T2;p:T3):T1
 is do end
end
class T1
end
class T2
end
class T3
end
```

deferred in the superclass. The new feature f has new formal arguments and also new return types.

In example 8 we listed the Eiffel representation of the transformed model by reverse engineering. In the next subsection, we will present the CTs which performed this transformation. First, we compute the set of exherited features to be created in the foster class and then we create the corresponding formal arguments and return types.

3.2.1 Computing the Exherited Feature Set

In order to compute the exherited feature set we use the CT from example 9. Each rule will be explained in details in the next paragraphs. First we compute the candidate feature set which contain tuples of features from the exherited classes having the same final name in the foster class. After this step the feature signatures are computed and types from corresponding positions are grouped together in order to compare them. Thus, we can decide whether they are compatible and if so, what is the representing type for the new feature in the foster class[2]. In our representation,

[2]In the case of anchored or generic types, structurally different types may be compatible, so the resulted type must be computed and added into the factbase.

Example 9 Computing the Exherited Feature Set

```
ct(
 createExheritedFeaturesInFosterClass(FosterClassId),
 condition((
  computeCandidateFeatures(FosterClassId,FeatureName,ExheritedFeatureIdList),
  computeSelectedFeatures(FosterClassId,ExheritedFeatureIdList,SelectedFeatureIdList),
  concatenateFeatureSignatureTypeLists(SelectedFeatureIdList,TypeIdLists),
  createTypeIdListCorrespondence(TypeIdLists,CorrespondentTypeIdLists),
  maplist(selectRepresentantTypeId,CorrespondentTypeIdLists,RepresentantTypeIdList),
  once(exists(featureBlock(FeatureBlockId,FosterClassId)))
)),
transformation((
 new_node_id(FeatureDeclId),
 new_node_id(RoutineId),
 add(featureDecl(FeatureDeclId,FeatureBlockId,FeatureName)),
 add(routine(RoutineId,FeatureDeclId)),
 add(deferredFeature(RoutineId))
))
).
```

signatures are implemented as lists of type identifiers, having on the first position the return type and then the list of formal argument types.

For each exherited feature we perform several actions. First the feature is created, then a routine is attached to it and finally the routine is set as deferred.

3.2.1.1 Computing the Candidate Features

The rule from example 10 computes the candidate features having the same final name in the foster class.

In example 10 we search for sets of features in the exherited classes. There will be taken only one feature from each exherited class, which have the same final name to build a set. Such feature sets are candidates for exheritance. However, feature final name matching is not sufficient for exheritance, the exherited features require to have compatible signatures, too. These issues are discussed in the next subsection.

3.2.1.2 Computing the Selected Features

From the set of exheritable features, we keep only the features which are selected for exheritance by the programmer. The selection can be performed by keywords like **all**, **nothing**, **only**, **except**.

In the implementation, a Prolog rule called *isFeatureSelectedForExheritance* is defined for each selection keyword. First the *allFeature* fact existence is tested for the given foster class. Next, if the previous rule fails, the *onlyFeature* fact existence is tested with foster class and feature identifiers. If the previous rule fails, then, the non-existence of the *exceptFeature* fact is checked, having the feature and foster class identifiers as arguments. Finally, the non-existence of *nothingFeature* fact is verified on the selected foster class. If there are no selection facts used the non-existence of *nothingFeature* will allow the selection of all candidate features. However, only one selection keyword may be used at a time.

3.2.1.3 Concatenating Feature Signature Type Lists

After we found the sets of candidate features we concatenate their types in lists, resulting signatures. Such a signature may have one of the following forms:

Example 10 Computing Candidate Features

```
computeCandidateFeatures(FosterClassId,FeatureName,FeatureList):-
 findall(
  (FosterClassId,FeatureName,FeatureList),
  computeCandidateFeatures0(FosterClassId, FeatureName, FeatureList),
  All
 ),
 sort(All,Sorted),
 member((FosterClassId,FeatureName,FeatureList),Sorted).
computeCandidateFeatures0(FosterClassId,FeatureName,FeatureList):-
 exists(foster(FosterClassId)),
 exists(exheritance(_,FosterClassId,ExheritedClassTypeId)),
 exists(classType(ExheritedClassTypeId,ExheritedClassId)),
 exists(classDecl(ExheritedClassId,_,_,_)),
 exists(featureBlock(FeatureBlockId,ExheritedClassId)),
 exists(featureDecl(FeatureDeclId,FeatureBlockId,_)),
 exheritedFeatureFinalName(FeatureDeclId,FosterClassId,ExheritedClassId,FeatureName),
 createExheritedClassList(FosterClassId,ExheritedClassIdList),
 forall(member(ExheritedClassId1,ExheritedClassIdList),
  exheritedFeatureFinalName(_,FosterClassId,ExheritedClassId1,FeatureName)),
 findall(FeatureDeclId1,
  exheritedFeatureFinalName(FeatureDeclId1,FosterClassId,_,FeatureName),
  FeatureList).
```

Example 11 Computing the Selected Features

```
computeSelectedFeatures(FosterClassId,ExheritedFeatureIdList,
 SelectedFeatureIdList):-
 findall(FeatureId,
  (member(FeatureId,ExheritedFeatureIdList),
   isFeatureSelectedForExheritance(FeatureId,FosterClassId)),
  SelectedFeatureIdList).

isFeatureSelectedForExheritance(_,FosterClassId):-
 exists(allFeature(FosterClassId)),
 !.
isFeatureSelectedForExheritance(FeatureId,FosterClassId):-
 exists(onlyFeature(FosterClassId,_))->
 exists(onlyFeature(FosterClassId,FeatureId)),
 !.
isFeatureSelectedForExheritance(FeatureId,FosterClassId):-
 exists(exceptFeature(FosterClassId,_))->
 not(exists(exceptFeature(FosterClassId,FeatureId))),
 !.
isFeatureSelectedForExheritance(_,FosterClassId):-
 not(exists(nothingFeature(FosterClassId))).
```

```
01 [noReturnType, noFormalArguments]
02 [noReturnType, 101, 102, 103]
03 [101, NoFormalArguments]
04 [101, 101, 102, 103]
```

Line 01 contains a signature which has neither formal arguments nor return type. In line 02 we have a signature which has no return type but there are formal arguments having three type identifiers listed. In line 03 we have a return type on the first position but there are no normal arguments. Line 04 describes a signature which has return type *[101]* and three formal argument types *[101, 102, 103]*. We can notice that in our convention the return type identifier is on the first position in the list. Its absence is marked by a special atom named *noReturnType*. The formal arguments start on the list second position and take the next positions. The formal arguments absence is marked by the appearance of the *noFormalArguments* atom.

For example the signature of feature *f* from class *EC1* is represented by the following list *[1101, 1101, 1102, 1103]*. The first element is the type identifier of the of the return type, while the next three arguments are the identifiers of the formal argument types.

In example 12 we listed several rules dealing with feature signature manipulation.

The rule *createFeatureSignatureTypeIdList* takes as input parameter a feature identifier and returns its signature (the list of formal arguments and return types).

The rule *concatenateFeatureSignatureTypeLists* takes as input a list of features, computes for each feature its signature and returns a list of signatures. For example if feature *701* has as signature list *[101, 101, 102, 103]*, feature *702 [101, 101, 102, 104]* and feature *703 [101, 101, 102, 105]*. The result of the rule will be *[[101, 101, 102, 103], [101, 101, 102, 104], [101, 101, 102, 105]]*.

The last rule *createTypeIdListCorrespondence* groups corresponding types having the same position in the lists in order to be compared. The effect applied on the previously obtained list is: *[[101, 101, 101], [101, 101, 101], [102, 102, 102], [103, 104, 105]]*. The intention of this arrangement is to facilitate the computation of the resulting types for the exherited feature in the foster class. Type exheritance issues will be presented in section 3.3.

3.2.2 Creating Formal Arguments

After the features are created in the foster class, their arguments must be created also: argument names are copied from one of the exherited classes candidate feature (if there is a feature implementation selected by **moveup**, that feature will provide the formal argument names), while their types are computed using a special algorithm which will be presented later in section 3.3.

After the creation of exherited features in the foster class, we have to complete them with the creation of formal argument and return type facts. In example 13 the formal arguments are created using the same names from the feature in the selected exherited class. The created formal argument facts have no reference to the parent fact and they are listed in the argument list of the selected subclass.

In example 14 we create the formal argument list fact which will refer the formal arguments created in the previous step. Iterating the argument list from the exherited feature in the subclass we can locate and build the new argument list for the foster class feature.

The third step of argument update, illustrated in example 15, consists in updating the parent of all created formal arguments and the like type if it is the case. Some like types require special updates because like type exheritance requires that the feature exheritance process to be finished completely. These issued will be discussed in section 3.3.

3.2.3 Creating Return Types

After formal argument creation we create also the return types and we attach them to the newly created features.

Example 12 Concatenating Feature Signature Type Lists

```
createFeatureSignatureTypeIdList(FeatureDeclId,SignatureTypeIdList):-
 exists(featureDecl(FeatureDeclId,FeatureBlockId,_)),
 exists(featureBlock(FeatureBlockId,ExheritedClassId)),
 exists(classDecl(ExheritedClassId,_,_,_)),
 exists(classType(ExheritedClassTypeId,ExheritedClassId)),
 exists(exheritance(_,FosterClassId,ExheritedClassTypeId)),
 exists(foster(FosterClassId)),
 extractReturnTypeFromFeature(FeatureDeclId,ReturnTypeIdList),
 extractFormalArgumentTypesFromFeature(FeatureDeclId,
  FormalArgumentTypeIdList),
 append(ReturnTypeIdList,FormalArgumentTypeIdList,
 SignatureTypeIdList).
concatenateFeatureSignatureTypeLists([],[]).
concatenateFeatureSignatureTypeLists([FeatureDeclId|FeatureIdList],
 [SignatureTypeIdList|SignatureTypeIdLists]):-
 exists(featureDecl(FeatureDeclId,_,_)),
 createFeatureSignatureTypeIdList(FeatureDeclId,SignatureTypeIdList),
 concatenateFeatureSignatureTypeLists(FeatureIdList,SignatureTypeIdLists).
createTypeIdListCorrespondence(SignatureTypeIdLists,
 CorrespondingTypeIdLists):-
 maplist(length,SignatureTypeIdLists,LengthList),
 sort(LengthList,[MaxLength]),
 createTypeIdListCorrespondence0(SignatureTypeIdLists,1,MaxLength,
 CorrespondingTypeIdLists).
createTypeIdListCorrespondence0(_,Index1,Index2,[]):-
Index1 > Index2.
createTypeIdListCorrespondence0(SignatureTypeIdLists,
 Index1,Index2,[CorrespondingTypeIdList|CorrespondingTypeIdLists]):-
 Index1=<Index2,
 findall(TypeId,
 (member(SignatureTypeIdList,SignatureTypeIdLists),
 nth1(Index1,SignatureTypeIdList,TypeId)),
 CorrespondingTypeIdList),
 Index11 is Index1+1,
 createTypeIdListCorrespondence0(SignatureTypeIdLists,
 Index11,Index2,CorrespondingTypeIdLists).
```

Example 13 Creating Formal Arguments

```
ct(
createExheritedFeatureArgumentsInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,ExheritedFeatureIdList,[_|RepresentingTypeIdList]),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 selectFeatureForFormalArgumentExheritance(FeatureDeclId2,ExheritedFeatureIdList),
 exists(formalArguments(FormalArguments2,FeatureDeclId2,FormalArgumentIdList2)),
 member(FormalArgumentId2,FormalArgumentIdList2),
 nth1(Index,FormalArgumentIdList2,FormalArgumentId2),
 nth1(Index,RepresentingTypeIdList,RepresentingTypeId3),
 exists(formalArgument(FormalArgumentId2,FormalArguments2,FormalArgumentName2,_))
)),
transformation((
 new_node_id(FormalArgumentId3),
 add(formalArgument(FormalArgumentId3,FormalArguments2,FormalArgumentName2,
 RepresentingTypeId3))
))
).
```

Example 14 Creating Formal Argument List

```
ct(
createExheritedFeatureArgumentListInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,ExheritedFeatureIdList),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 selectFeatureForFormalArgumentExheritance(FeatureDeclId2,ExheritedFeatureIdList),
 exists(formalArguments(FormalArgumentsId2,FeatureDeclId2,FormalArgumentIdList2)),
 findall(FormalArgumentId3,
 (
  member(FormalArgumentId2,FormalArgumentIdList2),
  exists(formalArgument(FormalArgumentId2,FormalArgumentsId2,FormalArgumentName,_)),
  exists(formalArgument(FormalArgumentId3,FormalArgumentsId2,FormalArgumentName,_)),
  FormalArgumentId2=\=FormalArgumentId3
 ),
 FormalArgumentIdList)
)),
transformation((
 new_node_id(FormalArgumentsId),
 add(formalArguments(FormalArgumentsId,ExheritedFeatureDeclId,FormalArgumentIdList))
))
).
```

Example 15 Updating Formal Arguments

```
ct(
updateExheritedFeatureArgumentsInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,CorrespondentTypeIdLists),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 exists(formalArguments(FormalArgumentsId,ExheritedFeatureDeclId,
  FormalArgumentIdList)),
 member(FormalArgumentId,FormalArgumentIdList),
 nth1(Index,FormalArgumentIdList,FormalArgumentId),
 nth0(Index,CorrespondentTypeIdLists,TypeIdList),
 exists(formalArgument(FormalArgumentId,_,FormalArgumentName,
  FormalArgumentTypeId)),
 computeLikeType(TypeIdList,FormalArgumentTypeId,FormalArgumentTypeId2)
)),
transformation((
 delete(formalArgument(FormalArgumentId,_,_,_)),
 add(formalArgument(FormalArgumentId,FormalArgumentsId,FormalArgumentName,
 FormalArgumentTypeId2))
))
).
```

Example 16 Creating Return Types

```
ct(
createExheritedFeatureTypeMarkInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,[TypeIdList|_],[TypeId|_]),
condition((
 exists(featureBlock(FeatureBlockId,FosterClassId)),
 exists(featureDecl(ExheritedFeatureDeclId,FeatureBlockId,_)),
 computeLikeType(TypeIdList,TypeId,TypeId2),
 TypeId2\==noReturnType
)),
transformation(
 add(typeMark(ExheritedFeatureDeclId,TypeId2))
)
).
```

Example 17 Exheriting Types Having The Same Identifier

```
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 sort(TypeIdList,[RepresentantTypeId]),
 !.
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 maplist(getLikeCurrentType,TypeIdList,LikeCurrentTypeIdList),
 sort(LikeCurrentTypeIdList,[RepresentantTypeId]),
 !.
getLikeCurrentType((TypeId,_),TypeId).
```

In the fourth step of the process (example 16) there are created facts for the return types. From the signature fact we extract the first type in the list and we create a type mark fact, if needed.

3.3 Type Exheritance

In this section we discuss how we select the representing type from a list of correspondent types from the exherited classes in several contexts. We call this action type exheritance. The simplest case of type exheritance occurs when all correspondent types have the same identifiers, meaning that they denote the same type[3]. The representing type is the unique type identifier. The types we consider are all the types from the Eiffel type system and they are: class types referring class declarations and having no actual arguments, separate and expanded type referring class types with the same restriction, like current types, bit types referring constants. Class types referring generic classes or having different but equivalent actual generics, like types having anchors, bit types referring constant features must be treated separately. For types having actual generics and different identifiers, the type exheritance process is recursive.

In example 17 we start from a type identifier list, we sort it in order to eliminate duplicates and finally it is expected to obtain a list having a single representing type identifier. For **like current** types, which depend very much on the class context they are used in, the type information is stored in a special structure having two elements *(TypeId, ClassDeclId)*. As we already know that they are **like current** types coming from subclasses, we are interested only in the first identifier *TypeId*, which has the same value for all **like current** types in the factbase. If none of the two rules matches the conditions, then the next rules for representing type computation will be used.

3.3.1 Exheriting Class Types Having Actual Arguments

In this subsection we analyze class type exheritance when types have no equal identifiers. This means that they might have actual arguments or that they are represented by formal generics. Here we deal only with class types having actual arguments. The candidate type selection is made in two steps in order to have a better reuse of the rules: first the *classType* facts referred by the *type* facts are extracted (example 18) and then the representing type is computed (example 19).

In example 18 we show how the *classType* facts are extracted from the *type* facts and how the type selection rule is called. The only constraint invoked by this rule is that the base class of all types should be identical. The generic class which is instantiated is known in Eiffel terminology as base class. This operation is performed by the sorting rule which eliminates the duplicates. Afterwards, only one element should remain in the list.

In example 19 the representing type is computed in the following way:

- actual arguments are extracted, they represent the types which instantiate the generic class;

[3]Each type is referred in the factbase by a unique identifier.

Example 18 Exheriting Class Types Having Actual Arguments (1)

```
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 maplist(extractClassDeclFromClassType,TypeIdList,ClassDeclIdList),
 sort(ClassDeclIdList,[_]),
 maplist(extractClassTypeFromType,TypeIdList,ClassTypeIdList),
 selectRepresentantTypeId(ClassTypeIdList,RepresentantTypeId),
 !.
extractClassTypeFromType(TypeId,ClassTypeId):-
 exists(type(TypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)).
```

- the actual arguments are reorganized in lists of types based on position criteria (the very same principle was used for feature exheritance);
- for those lists the representing type is computed recursively;
- the resulted type is assembled to represent the final result.

The *selectRepresentantTypeId* rule does all the steps iterated: it uses *extractActualGenericTypeIdListFromClassType* in order to extract the type list representing the actual arguments, creates the correspondence list by calling the *createTypeIdListCorrespondence* rule, recursively computes the representing types for the actuals by calling the *selectRepresentantTypeId* rule, and finally creates a class type with the resulted actuals. We mention also that the actual types may be any kind of Eiffel types and we apply the same rules for them as we apply in the general process of type exheritance.

3.3.2 Exheriting Expanded Types

In Eiffel variables of expanded types represent objects and not references to those objects. When such a type appears in the exherited features signatures then all the correspondent ones must be the same expanded type in order to be exherited.

In example 20 all types from the *ExpandedTypeIdList* parameter are checked to be expanded types by the rule *isTypeReferringExpandedType*. In the metamodel we designed an expanded type as refering to a class type. So, in this rule we detached the class types in a separate list through *extractClassTypeFromExpandedType* and we apply recursively the *selectRepresentantTypeId* rule. The result will be a class type which will create the resulted expanded type by the *createExpandedTypeFromClassType* rule.

3.3.3 Exheriting Separate Types

The **separate** keyword introduces the concept of inter-object concurrency in the Simple Concurent Object-Oriented Programming (SCOOP) mechanism of Eiffel [FOP04]. A class declared as separate means that the class executes its own thread of control. Separate arguments mean that they are synchronization points among concurrent threads. To exherit a feature having a separate type it implies that all corresponding types must be separate and must point to the same class. The resulted type is the same separate type present in the correspondence list.

The implementation provided is very similar to the one used with expanded types. In example 21 all types from the *SeparateTypeIdList* parameter are checked to be separate types by the rule *isTypeReferringSeparateType*. In the metamodel we designed a separate type as refering to a class type. So, in this rule we detached the class types in a separate list through *extractClassTypeFromSeparateType* and we apply recursively the *selectRepresentantTypeId* rule. The result will be a class type which will create the resulted separate type by the *createSeparateTypeFromClassType* rule.

Example 19 Exheriting Class Types Having Actual Arguments (2)

```
selectRepresentantTypeId(ClassTypeIdList,RepresentantTypeId):-
 member(ClassTypeId,ClassTypeIdList),
 exists(classType(ClassTypeId,ClassDeclId)),
 exists(classDecl(ClassDeclId,_,_,_)),
 !,
 maplist(extractActualGenericTypeIdListFromClassType,ClassTypeIdList,
  ActualGenericsTypeIdLists),
 createTypeIdListCorrespondence(ActualGenericsTypeIdLists,
  CorrespondentActualGenericsTypeIdLists),
 maplist(selectRepresentantTypeId,CorrespondentActualGenericsTypeIdLists,
  ActualGenericsRepresentantTypeIdList),
 createClassTypeWithActualGenericTypes(ClassDeclId,
  ActualGenericsRepresentantTypeIdList,RepresentantTypeId),
 !.
extractActualGenericTypeIdListFromClassType(ClassTypeId,TypeIdList):-
 exists(classType(ClassTypeId,ClassDeclId)),
 exists(classDecl(ClassDeclId,_,_,FormalGenericIdList)),
 length(FormalGenericIdList,Length),
 Length=\=0,
 extractActualGenericTypeIdListFromClassType0(ClassTypeId,ClassDeclId,
  FormalGenericIdList,TypeIdList),
 !.
extractActualGenericTypeIdListFromClassType(_,[]).
extractActualGenericTypeIdListFromClassType0(_,_,[],[]).
extractActualGenericTypeIdListFromClassType0(ClassTypeId,ClassDeclId,
 [FormalGenericId|FormalGenericIdList],
 [TypeId|TypeIdList]):-
 exists(formalGeneric(FormalGenericId,ClassDeclId,_)),
 exists(actualGenericType(_,ClassTypeId,FormalGenericId,TypeId)),
 extractActualGenericTypeIdListFromClassType0(ClassTypeId,ClassDeclId,
  FormalGenericIdList,TypeIdList).
```

Example 20 Exheriting Expanded Types

```
selectRepresentantTypeId(ExpandedTypeIdList,RepresentantTypeId):-
 maplist(isTypeReferringExpandedType,ExpandedTypeIdList,ResultList),
 sort(ResultList,['yes']),
 maplist(extractClassTypeFromExpandedType,ExpandedTypeIdList,ClassTypeIdList),
 selectRepresentantTypeId(ClassTypeIdList,TypeId),
 createExpandedTypeFromClassType(TypeId,RepresentantTypeId),
 !.
extractClassTypeFromExpandedType(TypeId,ClassTypeId):-
 exists(type(TypeId,ExpandedTypeId)),
 exists(expandedType(ExpandedTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringExpandedType(TypeId,'yes'):-
 exists(type(TypeId,ExpandedTypeId)),
 exists(expandedType(ExpandedTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringExpandedType(_,'no').
```

Example 21 Exheriting Separate Types

```
selectRepresentantTypeId(SeparateTypeIdList,RepresentantTypeId):-
 maplist(isTypeReferringSeparateType,SeparateTypeIdList,ResultList),
 sort(ResultList,[yes]),
 maplist(extractClassTypeFromSeparateType,SeparateTypeIdList,ClassTypeIdList),
 selectRepresentantTypeId(ClassTypeIdList,TypeId),
 createSeparateTypeFromClassType(TypeId,RepresentantTypeId),
 !.
extractClassTypeFromSeparateType(TypeId,ClassTypeId):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringSeparateType(TypeId,'yes'):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
isTypeReferringSeparateType(_,'no').
 extractClassTypeFromSeparateType(TypeId,ClassTypeId):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
```

Example 22 Exheriting Like Types (1)

```
selectRepresentingTypeId(TypeIdList,(RepresentingTypeId,ScheduledTypeList)):-
 maplist(computeTypeFinalType,TypeIdList,FinalTypeIdList),
 TypeIdList\==FinalTypeIdList,
 selectRepresentingTypeId(FinalTypeIdList,(RepresentingTypeId,
 ScheduledTypeList)),!.
```

3.3.4 Exheriting Like Types

Like types are special types which refer the type of a feature, formal argument or current class denoted by the **current** keyword. Such a type can be defined recursively involving a series of features or formal arguments. For example the type of a feature f can be like g and the type of g can be like h and so on, until the final feature has a non-anchored type. In order to compute the final type of an anchored type we have to call recursively several rules specialised in computing the final type for each kind of Eiffel type.

In example 22 the representing type is obtained after computing the final type for each component from the type list. If the two type sets, original and computed, are not equal that means that there were some anchored types or instantiated generics for which type inference was executed. Unless this check is performed the representing type computing rule will go on an infinite recursion. So, next the representing type is computed using the rules defined in the previous subsections.

For the non-anchored types of Eiffel we defined the final type to be the type itself (see example 23). The only exception for these rules are the class types refering formal generic instantiated with some concrete types.

In example 24, for like types the final type may be computed immediately being either the type of a feature, formal argument or **current** or recursively if there is a chain of referred entities.

3.3.5 Exheriting Bit Types

Bit types are used in Eiffel to represent integers of different sizes. Bit types may be expressed using a manifest constant or an integer constant feature. No matter how the candidate bit types are expressed the result type must be a bit type of the same size. Even if the constant features referred by the bit types are exheritable, they can not be redefined in the subclasses again as constant features. The chosen solution is to exherit the bit type using a manifest constant no matter of its origins in the subclasses. Another possible solution which affects more the exherited classes is to move up the constant feature. Such a solution can be triggered by selecting explicitly the implementation exheritance.

In example 25 the type representing selection rule checks all types to have the same size using the *maplist* operator with *getBitTypeValue* rule on the type list. The resulted list is sorted, duplicates are eliminated and a single integer is expected to remain. Afterwards, a bit type is created by the *createBitType* rule if the type does not already exist.

3.4 Exheriting Features with Rename Clauses

In this section we analyse a use case where features in foster classes have different names and they are exherited in the foster class under a common name. To do this, the exherited features are renamed in the foster class, using a common name. The aim of the experiment is to show that it is possible to generate semantically equivalent classes in the case of renaming.

The renaming in the context of reverse inheritance is linked to the exheritance branch having the same symmetrical behavior like ordinary inheritance which is linked to the inheritance branch. In exheritance like in inheritance some feature may be renamed and some may not. So we address

Example 23 Exheriting Like Types (2)

```
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,ClassTypeId)),
 exists(classType(ClassTypeId,ClassDeclId)),
 exists(classDecl(ClassDeclId,_,_,_)),
 !.
computeTypeFinalType(TypeId,FinalTypeId):-
 exists(type(TypeId,ClassTypeId)),
 exists(classType(ClassTypeId,FormalGenericId)),
 exists(formalGeneric(FormalGenericId,_,_)),
 exists(classDecl(ExheritedClassId,_,_,FormalGenericIdList)),
 memberchk(FormalGenericId,FormalGenericIdList),
 exists(classType(ExheritedClassTypeId,ExheritedClassId)),
 exists(exheritance(_,_,ExheritedClassTypeId)),
 exists(actualGenericType(_,ExheritedClassTypeId,FormalGenericId,
  FinalTypeId)),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,ExpandedTypeId)),
 exists(expandedType(ExpandedTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,SeparateTypeId)),
 exists(separateType(SeparateTypeId,ClassTypeId)),
 exists(classType(ClassTypeId,_)),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,BitTypeId)),
 exists(bitType(BitTypeId,_)).
```

Example 24 Exheriting Like Types (3)

```
computeTypeFinalType((TypeId,ClassDeclId),RepresentingTypeId):-
 exists(type(TypeId,LikeTypeId)),
 exists(likeType(LikeTypeId,IdentifierId)),
 exists(identifier(IdentifierId,'current')),
 createClassTypeFromClassDecl(ClassDeclId,RepresentingTypeId),
 !.
computeTypeFinalType(TypeId,TypeId):-
 exists(type(TypeId,LikeTypeId)),
 exists(likeType(LikeTypeId,IdentifierId)),
 exists(identifier(IdentifierId,'current')),
 !.
computeTypeFinalType(TypeId,FinalTypeId):-
 exists(type(TypeId,LikeTypeId)),
 exists(likeType(LikeTypeId,Id)),
 ((exists(featureDecl(Id,_,_)),
   exists(typeMark(Id,TempTypeId)));
   exists(formalArgument(Id,_,_,TempTypeId))),
   exists(type(TempTypeId,_)),
 computeTypeFinalType(TempTypeId,FinalTypeId),!.
```

Example 25 Exheriting Bit Types

```
selectRepresentantTypeId(TypeIdList,RepresentantTypeId):-
 maplist(getBitTypeValue,TypeIdList,ResultSizeList),
 sort(ResultSizeList,[IntegerValue]),
 integer(IntegerValue),
 createBitType(IntegerValue,RepresentantTypeId),
 !.
getBitTypeValue(TypeId,IntegerValue):-
 exists(type(TypeId,BitTypeId)),
 exists(bitType(BitTypeId,XId)),
 ((exists(featureDecl(XId,_,_)),
  exists(featureManifestConstant(XId,ManifestConstantId)),
  exists(manifestConstant(ManifestConstantId,IntegerValue,'integer')));
  (exists(manifestConstant(XId,IntegerValue,'integer')))).
```

only those features which are affected by renaming and we do not touch the ones which are not affected.

In example 26 we have a configuration of four classes one foster class *FC* and three exherited classes *EC1*, *EC2*, *EC3*. In each class there is a feature having different names: in *EC1* there is a feature named *f*, in *EC2* there is a feature *g* and in *EC3* is a feature *h*. All three features have the same semantics and it is indented to be exherited as f in the foster class. This is why in the foster class on the *EC2* and *EC3* exheritance branches there are placed two renaming clauses making the exherited features to have the same common name.

In the model from example 27 the only new clauses that appear are the rename clauses. The *rename* clause is linked by identifiers to the exheritance branch and to the renamed feature. It also contains the new name of the feature.

After model transformations we obtain the following clauses listed in example 28. In the model we notice that we have a deferred feature *f* in the foster class, so the feature was exherited correctly. Also we can notice that we have two rename clauses in the *EC2* and *EC3* subclasses. The rename clauses rename the feature from the superclass from its name to the original names of the features in the subclasses: *f* is renamed as *g* in *EC2* and *f* is renamed as *h* in *EC3*.

In the final code from example 40 we can see once again from the Eiffel perspective that the generated code do not affect the semantics of the original classes. Using renaming we solved the problem of inheriting the unwanted features from the superclass. Next, we present the CT which made possible such a transformation.

3.4.1 Moving Renaming Clauses from Foster Classes Into Exherited Classes

In the CT from example 30 iterates all the exherited features, removes the rename clauses from the foster class and replaces them with equivalent renaming clauses in the exherited classes. For a particular exherited feature, the renaming clause from the foster class is deleted, then a new renaming clause is added into the corresponding exherited class, which will rename back the feature to its original name from the exherited class.

3.5 Exheriting Features with Redefine Clauses

In this section we treat the redefine semantics of reverse inheritance and we show how the transformed code must be adjusted in order to handle redefinitions. We remind the reader that redefinitions are used in inheritance and exheritance if the signatures are covariant. Covariance is the property of the two signatures, one in the superclass and the other in the subclass, of the same

Example 26 Exheriting Features with Rename Clauses (Original Eiffel Code)

```
foster class FC
 exherit
  EC1
  EC2
   rename g as f
  end
  EC3
   rename h as f
  end
  all
end
class EC1
 feature
  f is do end
end
class EC2
 feature
  g is do end
end
class EC3
 feature
  h is do end
end
```

Example 27 Exheriting Features with Rename Clauses (Original Prolog Clauses)

```
exists(cluster(1,'default')).
exists(classDecl(100,1,'FC',[])).   exists(classDecl(101,1,'EC1',[])).
exists(classDecl(102,1,'EC2',[])).  exists(classDecl(103,1,'EC3',[])).
exists(foster(100)).
exists(exheritance(501,100,101)).   exists(exheritance(502,100,102)).
exists(exheritance(503,100,103)).   exists(allFeature(100)).
exists(rename(652,502,712,'f')).    exists(rename(653,503,713,'f')).
exists(featureBlock(701,101)).      exists(featureBlock(702,102)).
exists(featureBlock(703,103)).
exists(featureDecl(711,701,'f')).   exists(featureDecl(712,702,'g')).
exists(featureDecl(713,703,'h')).
exists(routine(751,711)).           exists(routine(752,712)).
exists(routine(753,713)).
exists(compound(801,751,[])).       exists(compound(802,752,[])).
exists(compound(803,753,[])).
```

Example 28 Exheriting Features with Rename Clauses (Transformed Prolog Clauses)

```
exists(cluster(1, default)).
exists(classDecl(100, 1, 'FC', [])).    exists(classDecl(101, 1, 'EC1', [])).
exists(classDecl(102, 1, 'EC2', [])).   exists(classDecl(103, 1, 'EC3', [])).
exists(featureBlock(701, 101)).         exists(featureBlock(702, 102)).
exists(featureBlock(703, 103)).
exists(featureDecl(711, 701, f)).       exists(featureDecl(712, 702, g)).
exists(featureDecl(713, 703, h)).
exists(routine(751, 711)).              exists(routine(752, 712)).
exists(routine(753, 713)).
exists(compound(801, 751, [])).         exists(compound(802, 752, [])).
exists(compound(803, 753, [])).
exists(featureBlock(1086, 100)).        exists(featureDecl(1087, 1086, f)).
exists(routine(1088, 1087)).            exists(deferredFeature(1088)).
exists(rename(1089, 502, 1087, g)).     exists(rename(1090, 503, 1087, h)).
exists(inheritance(501, 101, 100)).     exists(inheritance(502, 102, 100)).
exists(inheritance(503, 103, 100)).     exists(deferredClass(100)).
```

Example 29 Exheriting Features with Rename Clauses (Transformed Eiffel Code)

```
deferred class FC
 feature
  f is deferred end
end
class EC1
 inherit FC
 feature
  f is do end
end
class EC2
 inherit
  FC
   rename f as g
  end
 feature
  g is do end
end
class EC3
 inherit
  FC
   rename f as h
  end
 feature
  h is do end
end
```

Example 30 Moving Renaming Clauses from Foster Classes Into Exherited Classes

```
ct(
 moveRenameFromFosterClassToExheritedClass(FosterClassId),
 condition((
  exists(classDecl(FosterClassId,_,_,_)),
  exists(foster(FosterClassId)),
  exists(exheritance(ExheritanceId,FosterClassId,ExheritedClassTypeId)),
  exists(classType(ExheritedClassTypeId,ExheritedClassId)),
  exists(classDecl(ExheritedClassId,_,_,_)),
  not(exists(foster(ExheritedClassId))),
  exists(rename(RenameId1,ExheritanceId,FeatureId2,FeatureName1)),
  exists(featureBlock(FeatureBlockId2,ExheritedClassId)),
  exists(featureDecl(FeatureId2,FeatureBlockId2,FeatureName2)),
  exists(featureBlock(FeatureBlockId1,FosterClassId)),
  exists(featureDecl(FeatureId1,FeatureBlockId1,FeatureName1))
 )),
 transformation((
  % remove the rename clause from the superclass
  delete(rename(RenameId1,ExheritanceId,FeatureId2,FeatureName1)),
  % adds a new rename in the subclass, so the behaviour is unchanged
  new_node_id(RenameId2),
  add(rename(RenameId2,ExheritanceId,FeatureId1,FeatureName2))
 ))
).
```

inherited feature that whose arguments and return types vary in the same direction: arguments and return types from signature in superclass are supertypes for the arguments and return types of the signature from the subclass. A redefined feature declaration in reverse inheritance is global for all exheritance branches, while in ordinary inheritance it has to be declared on each exheritance branch, if needed.

In our case study we declared a redefined feature in the foster class and we will show how the equivalent class hierarchy will handle the feature redefinition.

In the Eiffel code from example 31 we created a feature f in the foster class *FC*. There it is declared as a redefinition of features f from exherited classes *EC1*, *EC2* and *EC3*. Also the signature of f from *FC* is using type T while the correspondent features from subclasses use *T1*, *T2*, *T3*.

In the model from example 32 the redefinition of feature f from the foster class is made by the three *redefineExherit* clauses refering features *711*, *712*, *713*.

In the transformed Prolog clauses from example 33 there are three *redefineInherit* clauses, each one is linked to one inheritance branch *501*, *502*, *503*.

Finally in the transformed code from example 34 we can see the final class hierarchy. The *redefineExherit* clauses are now transformed into several *redefineInherit* clauses. Next, we present the two CTs which made possible such a transformation: one adding redefine clauses in exherited classes and the second removing the redefine clauses from foster classes.

3.5.1 Adding Redefine Clauses to the Exherited Classes

In example 35 we present a CT, which on all inheritance branches will add a *redefineInherit* clause for a newly created feature. The newly created features are located using the temporary fact *candidateToNewFeatureMap* which has two identifier arguments: the one of a candidate feature and the one of the newly created feature.

In example 36 we present a CT which on all inheritance branches will change the *redefineExherit*

Example 31 Exheriting Features with Redefine Clauses (Original Eiffel Code)

```
foster class FC
 exherit
  EC1
  EC2
  EC3
 all
 redefine f
 feature f:T is do end
end
class EC1
 feature f:T1 is do end
end
class EC2
 feature f:T2 is do end
end
class EC3
 feature f:T3 is do end
end
class T end
class T1 inherit T end
class T2 inherit T end
class T3 inherit T end
```

Example 32 Exheriting Features with Redefine Clauses (Original Prolog Clauses)

```
exists(cluster(1,'default')).
exists(classDecl(100,1,'FC',[])).      exists(classDecl(101,1,'EC1',[])).
exists(classDecl(102,1,'EC2',[])).     exists(classDecl(103,1,'EC3',[])).
exists(foster(100)).                   exists(exheritance(501,100,101)).
exists(exheritance(502,100,102)).      exists(exheritance(503,100,103)).
exists(allFeature(100)).               exists(redefineExherit(651,100,711)).
exists(redefineExherit(652,100,712)). exists(redefineExherit(653,100,713)).
exists(featureBlock(700,100)).         exists(featureBlock(701,101)).
exists(featureBlock(702,102)).         exists(featureBlock(703,103)).
exists(featureDecl(710,700,'f')).      exists(featureDecl(711,701,'f')).
exists(featureDecl(712,702,'f')).      exists(featureDecl(713,703,'f')).
exists(typeMark(710,1100)).            exists(typeMark(711,1101)).
exists(typeMark(712,1102)).            exists(typeMark(713,1103)).
exists(routine(750,710)).              exists(routine(751,711)).
exists(routine(752,712)).              exists(routine(753,713)).
exists(compound(800,750,[])).          exists(compound(801,751,[])).
exists(compound(802,752,[])).          exists(compound(803,753,[])).
exists(type(1100,1200)).               exists(type(1101,1201)).
exists(type(1102,1202)).               exists(type(1103,1203)).
exists(classType(1200,1300)).          exists(classType(1201,1301)).
exists(classType(1202,1302)).          exists(classType(1203,1303)).
exists(classDecl(1300,1,'T',[])).      exists(classDecl(1301,1,'T1',[])).
exists(classDecl(1302,1,'T2',[])).     exists(classDecl(1303,1,'T3',[])).
exists(inheritance(511,1301,1300)).    exists(inheritance(512,1302,1300)).
exists(inheritance(513,1303,1300)).
```

Example 33 Exheriting Features with Redefine Clauses (Transformed Prolog Clauses)

```
exists(cluster(1, default)).
exists(classDecl(100, 1, 'FC', [])).      exists(classDecl(101, 1, 'EC1', [])).
exists(classDecl(102, 1, 'EC2', [])).     exists(classDecl(103, 1, 'EC3', [])).
exists(featureBlock(700, 100)).           exists(featureBlock(701, 101)).
exists(featureBlock(702, 102)).           exists(featureBlock(703, 103)).
exists(featureDecl(710, 700, f)).         exists(featureDecl(711, 701, f)).
exists(featureDecl(712, 702, f)).         exists(featureDecl(713, 703, f)).
exists(typeMark(710, 1100)).              exists(typeMark(711, 1101)).
exists(typeMark(712, 1102)).              exists(typeMark(713, 1103)).
exists(routine(750, 710)).                exists(routine(751, 711)).
exists(routine(752, 712)).                exists(routine(753, 713)).
exists(compound(800, 750, [])).           exists(compound(801, 751, [])).
exists(compound(802, 752, [])).           exists(compound(803, 753, [])).
exists(type(1100, 1200)).                 exists(type(1101, 1201)).
exists(type(1102, 1202)).                 exists(type(1103, 1203)).
exists(classType(1200, 1300)).            exists(classType(1201, 1301)).
exists(classType(1202, 1302)).            exists(classType(1203, 1303)).
exists(classDecl(1300, 1, 'T', [])).      exists(classDecl(1301, 1, 'T1', [])).
exists(classDecl(1302, 1, 'T2', [])).     exists(classDecl(1303, 1, 'T3', [])).
exists(inheritance(511, 1301, 1300)).     exists(inheritance(512, 1302, 1300)).
exists(inheritance(513, 1303, 1300)).     exists(inheritance(501, 101, 100)).
exists(inheritance(502, 102, 100)).       exists(inheritance(503, 103, 100)).
exists(redefineInherit(651, 501, 710)).   exists(redefineInherit(652, 502, 710)).
exists(redefineInherit(653, 503, 710)).
```

clause into *redefineInherit* clause for a feature. Because at the moment the transformations from exheritance branches to inheritance branches is not yet made, but we know that they will keep the same own identifier, we use the exheritance branches identifiers.

3.6 Transforming Class Relationships

First we start with a simple Eiffel class hierarchy as it can be seen in example 37. The code contains only three classes: two exherited classes and one foster class. There are also two exheritance links between the foster class and each exherited class. The exherited features are selected by **all** keyword. There are no features present in any class of this hierarchy.

The equivalent Prolog clauses are presented in example 38. The two models represent the same entities but using different representations: Eiffel code and Prolog clauses. We note that there are class description clauses for all the three classes present in the hierarchy. The foster class is marked as being foster by a fact. The two exheritance class relationships are expressed using two clauses. The exherited feature selection is expressed also by a fact.

After the transformation the resulted model can be seen in example 39. We note that the exheritance clauses are transformed now into inheritance clauses and that the foster fact is now removed.

Finally, the equivalent Eiffel code can be found in example 40. We see the class configuration corresponding to the transformed model.

To perform the transformations several CTs were used:

- *transformExheritToInherit*
- *addDeferredToClass*

Example 34 Exheriting Features with Redefine Clauses (Transformed Eiffel Code)

```
class FC
 feature
  f is do end
end
class EC1
 inherit
  FC
   redefine f
  end
 feature
  f is do end
end
class EC2
 inherit
  FC
   redefine f
  end
 feature
  f is do end
end
class EC3
 inherit
  FC
   redefine f
  end
 feature
  f is do end
end
class T end
class T1 inherit T end
class T2 inherit T end
class T3 inherit T end
```

Example 35 Adding Redefine Clauses to the Exherited Classes (1)

```
ct(
 addRedefineClausesForNewFeaturesInTheExheritedClasses(FosterClassId),
 condition((
  exists(exheritance(ExheritanceId,FosterClassId,ExheritedClassTypeId)),
  exists(classType(ExheritedClassTypeId,ExheritedClassId)),
  exists(featureBlock(FeatureBlockId1,ExheritedClassId)),
  exists(featureDecl(FeatureId1,FeatureBlockId1,_)),
  exists(candidateToNewFeatureMap(FeatureId1,FeatureId2)),
  not(exists(redefineInherit(_,ExheritanceId,FeatureId1)))
 )),
 transformation((
  new_node_id(RedefineInheritId1),
  add(redefineInherit(RedefineInheritId1,ExheritanceId,FeatureId2))
 ))
).
```

Example 36 Adding Redefine Clauses to the Exherited Classes (2)

```
ct(
addRedefineClausesForRedefinedFeaturesInTheExheritedClasses(FosterClassId),
 condition((
  exists(foster(FosterClassId)),
  exists(classDecl(FosterClassId,_,_,_)),
  exists(redefineExherit(RedefineExheritId,FosterClassId,FeatureId1)),
  exists(featureDecl(FeatureId1,FeatureBlockId1,_)),
  exists(featureBlock(FeatureBlockId1,ExheritedClassId)),
  exists(classType(ExheritedClassTypeId,ExheritedClassId)),
  exists(exheritance(ExheritanceId,FosterClassId,ExheritedClassTypeId)),
  exheritedFeatureFinalName(FeatureId1,FosterClassId,ExheritedClassId,
   FeatureFinalName),
  exists(featureBlock(FeatureBlockId2,FosterClassId)),
  exists(featureDecl(FeatureId2,FeatureBlockId2,FeatureFinalName)),
  not(exists(redefineInherit(_,ExheritanceId,FeatureId2)))
 )),
 transformation((
  delete(redefineExherit(RedefineExheritId,FosterClassId,FeatureId1)),
  add(redefineInherit(RedefineExheritId,ExheritanceId,FeatureId2))
 ))
).
```

Example 37 Transforming Class Relationships (Original Eiffel Code)

```
class FC
 exherit
  EC1
  EC2
 all
end
class EC1 end
class EC2 end
```

Example 38 Transforming Class Relationships (Original Prolog Clauses)

```
exists(project('sample0100','.','sample0100t','.')).
exists(cluster(1,'default')).
exists(classDecl(10,1,'FC',[])).
exists(classDecl(11,1,'EC1',[])).
exists(classDecl(12,1,'EC2',[])).
exists(foster(10)).
exists(exheritance(101,10,111)).
exists(exheritance(102,10,112)).
exists(allFeature(10)).
exists(classType(111,11)).
exists(classType(112,12)).
```

Example 39 Transforming Class Relationships (Transformed Prolog Clauses)

```
exists(project(sample0100, '.', sample0100t, '.')).
exists(cluster(1, default)).
exists(classDecl(10, 1, 'FC', [])).
exists(classDecl(11, 1, 'EC1', [])).
exists(classDecl(12, 1, 'EC2', [])).
exists(inheritance(1001, 11, 110)).
exists(inheritance(1002, 12, 110)).
exists(classType(110,10)).
```

Example 40 Transforming Class Relationships (Transformed Eiffel Code)

```
class FC
end
class EC1
 inherit FC
end
class EC2
 inherit FC
end
```

- *removeFoster*

These CTs will be explained in detail in the next subsections.

3.6.1 Transforming Exheritance Clauses Into Inheritance Clauses

In this subsection we will discuss about the CT presented in example 41, which performs a transformation on the factbase. For a given foster class it deletes all the exheritance class relationship and replaces them with inheritance class relationships.

The CT presented in example 41 iterates the exheritance clauses for a given foster class and applies a *delete* and an *add* operation. The exheritance clauses are deleted then a new inheritance clause is created instead. We preferred to use the same identifier in order to keep the consistency with the rest of the model elements. Otherwise changes are required to all the child elements which refer exheritance branches. There are not used any other Prolog rules but just the facts from the factbase model.

3.6.2 Adding the Deferred Keyword to a Class

In this example 42 we describe the CT which adds the **deferred** keyword to a class if in that class has at least one deferred feature.

The CT selects only the foster classes which have at least one exherited feature and for those classes the *deferredClass* fact is added.

3.6.3 Removing Foster Keyword from a Foster Class

A final transformation in the list consists in removing the foster keyword from the foster classes. This transformation should be the last in the chain of transformations.

The CT consists in detecting the foster classes and removing their attached **foster** clause. In order to do this, in the conditional part of the CT we set the conditions: to be a class declaration and to have the foster fact attached, next in the action section of the CT the delete action is applied on the found foster fact.

Example 41 ct(transformExheritIntoInherit)

```
ct(
 transformExheritIntoInherit(FosterClassId),
 condition((
  exists(classDecl(FosterClassId,_,_,_)),
  exists(foster(FosterClassId)),
  exists(exheritance(ExheritanceId,FosterClassId,ExheritedClassId)),
  exists(classDecl(ExheritedClassId,_,_,_)),
  not(exists(foster(ExheritedClassId)))
 )),
 transformation((
  delete(exheritance(_,FosterClassId,ExheritedClassId)),
  new_node_id(InheritanceId),
  add(inheritance(InheritanceId,ExheritedClassId,FosterClassId))
 ))
).
```

Example 42 ct(addDeferredToClass)

```
ct(
 addDeferredToClass(FosterClassId),
 condition((
  exists(classDecl(FosterClassId,_,_,_)),
  exists(foster(FosterClassId)),
  exists(featureBlock(FeatureBlockId,FosterClassId)),
  exists(featureDecl(FeatureId,FeatureBlockId,_)),
  exists(routine(RoutineId,FeatureId)),
  exists(deferredFeature(RoutineId)),
  !
 )),
 transformation(
  add(deferredClass(FosterClassId))
 )
).
```

Example 43 ct(removeFoster)

```
ct(
 removeFoster(FosterClassId),
 condition((
  exists(classDecl(FosterClassId,_,_,_)),
  exists(foster(FosterClassId))
 )),
 transformation(
  delete(foster(FosterClassId))
 )
).
```

3.7 CT Execution Flow

In figure 3.1 are presented how the CTs are linked together using the operators described in section 3.1. The CTs are organized in a tree structure which is traversed in preorder, the traversal representing their execution flow. During the preorder traversal, the CTs are executed, propagating data through their parameters and performing the necessary transformations on the factbase model.

In example 44 there is listed the Prolog code of CTs from figure 3.1 along with their parameters.

Next, we will present the preorder traversal sequence of the CTs, explaining also how the parameters are transmitted:

- *ct(createExheritedFeatureBlockInFosterClass(FosterClassId, FeatureBlockId))* - is the first in the sequence. It iterates all foster classes and creates one feature block in each one;
- *ct(createExheritedFeaturesInFosterClass(FosterClassId, FeatureBlockId, NewFeatureDeclId, _NewFeatureName, ExheritedFeatureIdList, CorrespondentTypeIdLists, _ComplexRepresentingTypeIdList, RepresentingTypeIdList, ScheduledTypeList1))* - creates the exherited features, without their signatures, in the foster class placing them in the previously created feature block. The new signatures are computed and stored in the *RepresentingTypeIdList*. Also some new types may be created during this process and hey are stored in *ScheduledTypeList1*, to be added later to the factbase.
- *ct(createCandidateToNewFeatureMap(NewFeatureDeclId, ExheritedFeatureIdList))* - creates a set of temporary facts representing a map from the candidate features to the newly added feature in the foster class;
- *ct(createTypesForNewFeatureInFosterClass(ScheduledTypeList1))* - creates new types computed during a previous CT and which are used in the signatures of the exherited features;
- *ct(createExheritedFeatureArgumentsInFosterClass(FosterClassId,FeatureBlockId, NewFeatureDeclId, ExheritedFeatureIdList, RepresentingTypeIdList))* - creates the formal arguments for the exherited features using the signature stored in *RepresentingTypeIdList*;
- *ct(createExheritedFeatureArgumentListInFosterClass(FosterClassId, FeatureBlockId, NewFeatureDeclId, ExheritedFeatureIdList))* - since formal arguments are modelled in section 2.1 as an ordered list of references, these lists are created by the current CT;
- *ct(updateExheritedFeatureArgumentsInFosterClass(FosterClassId, FeatureBlockId, NewFeatureDeclId, CorrespondentTypeIdLists, ScheduledTypeList2))* - the newly created formal arguments need to be updated to their recently created parent list;
- *ct(createTypesForNewFeatureInFosterClass(ScheduledTypeList2))* - in the process of formal argument creation, the like types are recomputed trying to exherit them along with their anchor if possible and new types may appear which are added to the factbase;
- *ct(createExheritedFeatureTypeMarkInFosterClass(FosterClassId, FeatureBlockId, NewFeatureDeclId, CorrespondentTypeIdLists, RepresentingTypeIdList, ScheduledTypeList3))* - the return types are created from the previously computed signatures;
- *ct(createTypesForNewFeatureInFosterClass(ScheduledTypeList3))* - as for formal arguments new types may appear in the context of anchors which are added to the factbase;
- *ct(moveRenameFromFosterClassToExheritedClass(FosterClassId))* - on the foster class all the renaming clauses are reversed and re-targeted to the exherited classes to rename back the features;
- *ct(addRedefineClausesForNewFeaturesInTheExheritedClasses(FosterClassId))* - for the newly created features redefinition clauses are added into the subclasses;

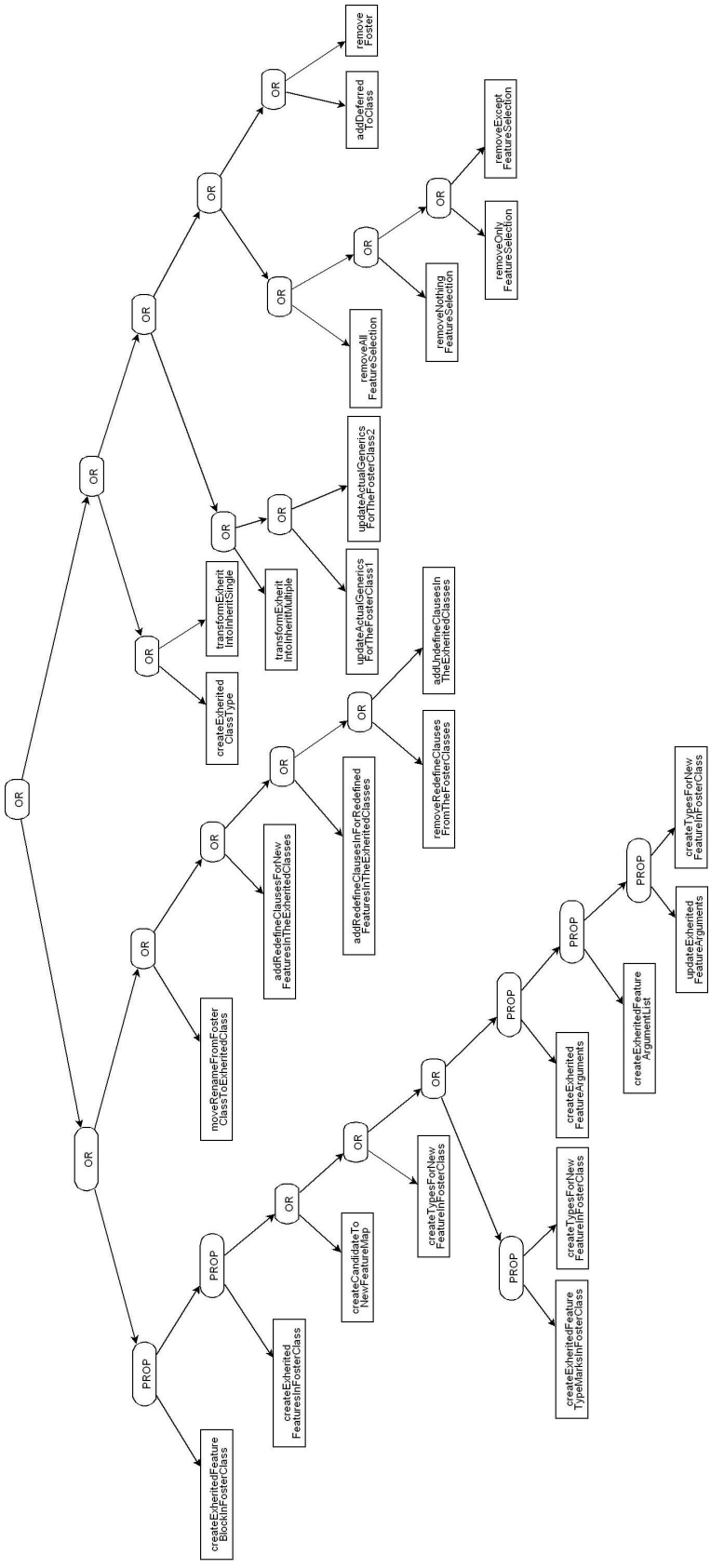

Figure 3.1: CT Tree

Example 44 CT Sequence

```
execCTS(
orseq(
orseq(
propseq(
ct(createExheritedFeatureBlockInFosterClass(FosterClassId,FeatureBlockId)),
propseq(ct(createExheritedFeaturesInFosterClass(FosterClassId,FeatureBlockId,
 NewFeatureDeclId,_NewFeatureName,ExheritedFeatureIdList,CorrespondentTypeIdLists,
 _ComplexRepresentingTypeIdList,RepresentingTypeIdList,ScheduledTypeList1)),
orseq(ct(createCandidateToNewFeatureMap(NewFeatureDeclId,ExheritedFeatureIdList)),
orseq(ct(createTypesForNewFeatureInFosterClass(ScheduledTypeList1)),
orseq(
propseq(ct(createExheritedFeatureArgumentsInFosterClass(FosterClassId,
 FeatureBlockId,NewFeatureDeclId,ExheritedFeatureIdList,RepresentingTypeIdList)),
propseq(ct(createExheritedFeatureArgumentListInFosterClass(FosterClassId,
 FeatureBlockId,NewFeatureDeclId,ExheritedFeatureIdList)),
propseq(ct(updateExheritedFeatureArgumentsInFosterClass(FosterClassId,
 FeatureBlockId,NewFeatureDeclId,CorrespondentTypeIdLists,ScheduledTypeList2)),
ct(createTypesForNewFeatureInFosterClass(ScheduledTypeList2)))),
propseq(ct(createExheritedFeatureTypeMarkInFosterClass(FosterClassId,
 FeatureBlockId,NewFeatureDeclId,CorrespondentTypeIdLists,
 RepresentingTypeIdList,ScheduledTypeList3)),
ct(createTypesForNewFeatureInFosterClass(ScheduledTypeList3)))))))),
orseq(ct(moveRenameFromFosterClassToExheritedClass(FosterClassId)),
orseq(ct(addRedefineClausesForNewFeaturesInTheExheritedClasses(FosterClassId)),
orseq(ct(addRedefineClausesForRedefinedFeaturesInTheExheritedClasses(
 FosterClassId)),
orseq(ct(removeRedefineClausesFromTheFosterClass(FosterClassId)),
ct(addUndefineClausesInTheExheritedClasses(FosterClassId)))))))),
orseq(
orseq(ct(createFosterClassType(FosterClassId)),
ct(transformExheritIntoInheritSingle(FosterClassId))),
orseq(
orseq(ct(transformExheritIntoInheritMultiple(FosterClassId)),
orseq(ct(updateActualGenericsForTheFosterClassTypes1(FosterClassId)),
ct(updateActualGenericsForTheFosterClassTypes2(FosterClassId)))),
orseq(
orseq(ct(removeAllFeatureSelection(FosterClassId)),
orseq(ct(removeNothingFeatureSelection(FosterClassId)),
orseq(ct(removeOnlyFeatureSelection(FosterClassId)),
ct(removeExceptFeatureSelection(FosterClassId))))),
orseq(ct(addDeferredToClass(FosterClassId)),
ct(removeFoster(FosterClassId)))))))).
```

- *ct(addRedefineClausesForRedefinedFeaturesInTheExheritedClasses(FosterClassId))* - for the explicitly redefined features in exheritance, feature redefinition clauses are added also to the subclasses;
- *ct(removeRedefineClausesFromTheFosterClass(FosterClassId))* - all the redefinition clauses of exheritance are removed;
- *ct(addUndefineClausesInTheExheritedClasses(FosterClassId))* - for any deferred feature in the subclasses corresponding to a newly created effective feature in the foster class an undefinition clause is created on the exheritance branch;
- *ct(createFosterClassType(FosterClassId))* - in order to change exheritance links into inheritance links and when the exherited classes are not generic, one class type pointing to the foster class is created;
- *ct(transformExheritIntoInheritSingle(FosterClassId))* - replaces exheritance with inheritance branches using a single class type pointing to the foster class and keeping their identifiers. Also, the exherited class types are deleted from the factbase, since they are no longer needed.
- *ct(transformExheritIntoInheritMultiple(FosterClassId))* - replaces the exheritance clauses into inheritance clauses and each exherited class type is changed into an inherited class type. Such transformation is needed when exherited and foster classes are generic and each inheritance clause points to a specific foster class type instantiated with different actual generics.
- *ct(updateActualGenericsForTheFosterClassTypes1(FosterClassId))* - the actual generics for the foster class type are computed from the exherited class instantiation. In section 2.3 we validate only models having a one to one instantiation relation between formal generics of foster class and exherited classes.
- *ct(updateActualGenericsForTheFosterClassTypes2(FosterClassId))* - instantiates the foster class with a concrete type which is equal to the constrained type in the foster class and the constrained type from the exherited classes;
- *ct(removeAllFeatureSelection(FosterClassId))* - the all feature selection clause is removed from the given foster class;
- *ct(removeNothingFeatureSelection(FosterClassId))* - the nothing selection clause is removed from the given foster class;
- *ct(removeOnlyFeatureSelection(FosterClassId))* - the explicit selection clauses are removed from the given foster class;
- *ct(removeExceptFeatureSelection(FosterClassId))* - the implicit selection clauses are removed from the given foster class;
- *ct(addDeferredToClass(FosterClassId))* - a deferred modifier is added to the foster class if there is at least one deferred feature;
- *ct(removeFoster(FosterClassId))* - the foster modifier is removed finally from the superclass.

From the point of view of parameter propagation we discuss the following CT parameters:

- *FosterClassId* - is propagated from the first CT to the last one. The whole transformations row are performed sequentially on each foster class.
- *FeatureBlockId* - is the newly created feature block to store the exherited features. It is propagated from the block creation CT to the exherited feature creation CT and also to further CTs, but is not used anymore.

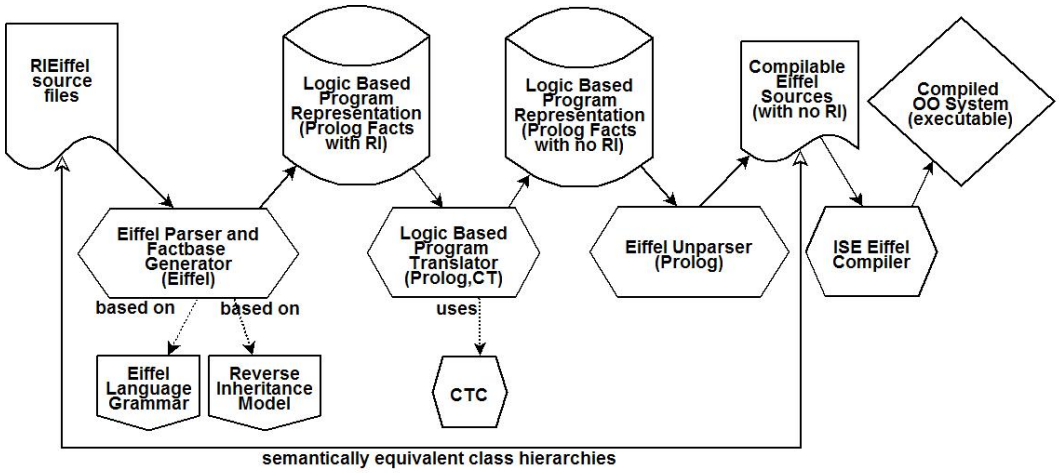

Figure 3.2: Software Instrumentation Overview

- *NewFeatureDeclId* - is the identifier of the newly created feature in the foster class;
- *_NewFeatureName* - is the name of the newly created feature;
- *ExheritedFeatureIdList* - is the list of the candidate features from the exherited classes ;
- *CorrespondentTypeIdLists* - is the list of the signature types grouped by entity;
- *_ComplexRepresentingTypeIdList* - is the list of types representing the resulting signature for the exherited feature and having also attached lists with new type facts to be added to the factbase;
- *RepresentingTypeIdList* - is the list of types representing strict the resulting signature for the exherited feature;
- *ScheduledTypeList1* - is the list of type facts to be added to the factbase.

3.8 Unimplemented Language Features

In the prototype some features defined in the RIEiffel were not implemented since they are not present in the GOBO library.

In the library, ordinary inheritance is not dual, it is only conforming, so reverse inheritance is implemented accordingly.

Tuples are not present, so they were not implemented in the context of reverse inheritance either.

Threads and concurrency mechanisms were not taken into account at all in the context of reverse inheritance, being out of the scope of the work.

3.9 Software Instrumentation

In this section we present the several software tools representing the prototype modules. The **RIEiffel** prototype is a **complete solution** for the implementation of the reverse inheritance class relationship. The prototype accepts as input a project re-engineered by reverse inheritance and produces the executable object-oriented system. The prototype is composed out of several modules, each corresponding to one phase of the proposed solution (see figure 3.2).

3.9.1 Prolog Factbase Generator

The **ETransformer** module converts the RIEiffel source files into a factbase stored as a Prolog file. The module is written in Eiffel and it was built from a generated parser. The parser was generated from the RIEiffel grammar listed in appendix A using the **gelex** (a version of Flex [Fou95] generating Eiffel code) and **geyacc** (a version of Yacc [Joh79] or Bison [Fou06] generating Eiffel code) tools from the **GOBO** library [Bez07]. The parsing process is applied on the whole Eiffel class universe including both client and system libraries classes. This is necessary since the analysis may require also information from the system library classes. The generated parser builds the abstract syntax tree (AST) of the input sources written in RIEiffel. Then the AST is visited and using a set of semantic actions, manually written, the Prolog facts are generated.

3.9.2 Prolog Factbase Transformer

All the CTs presented in this chapter are centralized in the **PTransformer** module of the prototype. The module takes at input the generated factbase and produces a new semantically equivalent factbase, which contains only pure Eiffel facts. The CTs are executed by the CTC interpretor [Kni06] which analyses them and performs the desired changes on the factbase.

The PTransformer module is tested automatically with two types of tests: structural and output. The structural tests consist in transforming a model samples and comparing the output against a model oracle. The output tests compares the regenerated Eiffel code for the transformed models against an output source oracle.

3.9.3 Eiffel Code Regenerator

The transformed factbase must be translated back into pure Eiffel code in order to obtain the executable object-oriented system. The **Unparser** module of the prototype is responsible for this task. The regeneration is done in Prolog by unparsing rules for each type of facts from the factbase following the RIEiffel[4] grammar.

The unparsing process consists in checking the existence of facts and printing at output the related keywords and values following the rules of RIEiffel grammar. In example 45 we present one code fragment from the **Unparser** module.

As an observation, one can notice that we regenerate both Eiffel and RIEiffel code, because we output the adaptation parts for a feature if they exist. Such a regeneration is necessary for prototype testing and integration. Thus can be tested the equivalence between the initial source files and the model, by comparing the source with the unparsed ones obtained from the model.

3.10 Summary

In this chapter we presented a set of transformations which express the semantics of reverse inheritance on the factbase model. We decided to use a declarative language, Prolog, since the semantics of reverse inheritance could be more easily expressed than using an imperative language based on AST (Abstract Syntax Tree) and Visitor design pattern [GHJV97]. The implementation is based on the CT abstraction which use parameters and operators to interact.

The exheritance of features is not a trivial process. First the candidate features are computed, next their signatures are compared. In this process the combination of all compatible types is taken into account. If we obtained successfully the list of types from the signature we can start creating the formal arguments and return types. During this creation process, like types are analysed again since there is a chance that the anchor was exherited and the new type could refer it. At this stage new type facts may appear which must be added to the factbase.

Next, the feature clauses are transformed in order to produce the equivalent code. Renamings are reversed, features from the subclasses are renamed back to their original names. Redefinition

[4]The grammar of RIEiffel includes the grammar of pure Eiffel. Any pure Eiffel source code will conform to the RIEiffel grammar.

Example 45 Unparsing Feature Declarations

```
unparseFeatureDecl(FeatureDeclId):-
 exists(featureDecl(FeatureDeclId,_,FeatureName)),
 tab(4),
 name(FeatureName,FeatureName2),
 writef('%s',[FeatureName2]),
 (exists(formalArguments(FormalArgumentsId,FeatureDeclId,_))->
  unparseFormalArguments(FormalArgumentsId);true),
 ((exists(typeMark(FeatureDeclId,TypeId))->
 (writef(':'),
  unparseType(TypeId)));true),
 ((exists(featureManifestConstant(FeatureDeclId,ManifestConstantId)),
   exists(manifestConstant(ManifestConstantId,Value,_))->
 (writef(' is %t',[Value])));true),
 ((exists(routine(RoutineId,FeatureDeclId))->
  unparseRoutine(RoutineId));nl),
 ((exists(attributeAdaptation(_,FeatureDeclId,_,_,_,_));
   exists(routineAdaptation(_,FeatureDeclId,_)))->
  unparseAdapted(FeatureDeclId);true).
```

clauses of exheritance are translated into equivalent inheritance redefinition clauses for both new features and redefined features in foster class.

The exheritance branches are transformed into inheritance branches using two strategies depending whether the exherited classes are generic or not. The actual generics used to instantiate the exherited classes will be reversed to instantiate the foster class. Finally, the feature selection mechanisms and the foster modifiers are deleted from the factbase.

After the description of all the CTs their tree organization is presented along with their operators and parameter data flow. Mainly there were used ORSEQ operators and some PROPSEQ operators. The main transformation flow is based on a foster class parameter. At the beginning of the transformation sequence some CTs propagate the newly created features in order to equip them with formal arguments and return types.

The three modules implementing the prototype are described. The first one is a RIEiffel parser which produces the Prolog facts. The second one is a translator written in Prolog and using the CT concept which generates the semantically equivalent model. The third module is an unparser of the Prolog factbase producing pure Eiffel code.

Chapter 4

Conclusions and Future Work

In the first chapter we presented several implementation possibilities for reverse inheritance. The adopted implementation solution does not invent a new programming language, but extends an existing one, so it resulted RIEiffel. The prototype is based on the GOBO Eiffel free library, thus it can be tested and used in industrial projects. As a consequence the grammar of GOBO was augmented with the reverse inheritance rules and was listed in appendix A. The grammar is designed incrementally so any pure Eiffel source file will conform to the extended grammar.

In the implementation the decision was taken to express the semantics of reverse inheritance through Prolog fact transformations. In chapter two we presented the metamodel for the logic representation of Eiffel programs. First the pure Eiffel entities and afterwards the reverse inheritance extension elements are modeled. The factbase metamodel respects the 3NF form for relational databases. The metamodel was enhanced with a set of Prolog rules responsible with the detection of semantically invalid factbases. For example the exherited feature selection clauses must select the features in an unique manner.

In chapter three are described in details the transformations expressed using the conditional transformation (CT) abstraction. The CTs from the transformation graph were designed to be executed sequentially in order to perform the necessary transformations. The *ANDSEQ*, *ORSEQ* and *PROPSEQ* operators were used to compose the CTs and thus to express the logic of the transformation. The order of some CTs is arbitrary. For example, the deletion of feature selection facts can be interchanged with the deletion of foster facts. Some CTs may not be even executed. For example, when exheriting a feature with no return types the CT responsible with the return type creation will not be executed. The transformations keep as much as necessary the reverse inheritance informations in the model and they are eliminated only at the end. This will help us better locate the model elements which need early transformations and avoid using new intermediary facts.

The transformed facts are translated back to pure Eiffel code using the Unparser module. The regenerated code is equivalent with the transformed model but will have a new text organization because of different text indentation. Code indentation information could be captured, but such details would complicate the model very much. Anyway, if a desired code formatting is required either the Unparser module is modified or another tool could be used.

The moveup mechanism implementation, assertion synthesis and adaptations implementation are currently under development.

As future work we must use the prototype against industrial strength Eiffel projects. Following this direction, we may experiment the reverse inheritance class relationship on the Eiffel Kernel Library. The library has a lot of classes modelling among others linked lists and array lists with exheritable common interfaces. Regenerating the code and recompiling the source code for such a library would represent quite a good scalability proof task for the RIEiffel prototype.

Another perspective for the reverse inheritance extension is to be integrated in the GOBO Eiffel library [Bez07]. We intend to design the integration schema to be as independent as possible of the GOBO library further evolution. The RIEiffel grammar files will override the standard ones

from the library, the AST files must be regenerated using the **gelex** and **geyacc** tools. The source files containing the classes modelling the Prolog facts can be copied in a new folder of the existing directory structure. Such an integration task could be easily automated by a shell script.

Appendix A

Eiffel Reverse Inheritance BNF Grammar Rules

```
Class_declaration: Indexing_opt Class_header Formal_generics_opt Obsolete_opt
Inheritance_opt Exheritance_opt Creators_opt Features_opt Invariant_opt E_END
Indexing_opt: -- /* empty */
| E_INDEXING Index_list
Index_list: -- /* empty */
| Index_clause
| Index_list Index_clause
| Index_list ';' Index_clause
Index_clause: Index_terms
| E_IDENTIFIER ':' Index_terms
Index_terms: Index_value
| Index_terms ',' Index_value
Index_value: E_IDENTIFIER
| Manifest_constant
Class_header: Header_mark_opt E_CLASS E_IDENTIFIER
Header_mark_opt: -- /* empty */
| E_DEFERRED [E_FOSTER]
| E_EXPANDED [E_FOSTER]
| E_SEPARATE
| E_FOSTER
Formal_generics_opt: -- /* empty */
| '[' Formal_generic_list ']'
Formal_generic_list: -- /* empty */
| E_IDENTIFIER Constraint_opt
| Formal_generic_list ',' E_IDENTIFIER Constraint_opt
Constraint_opt: -- /* empty */
| E_ARROW Class_type
Obsolete_opt: -- /* empty */
| E_OBSOLETE E_STRING
Inheritance_opt: -- /* empty */
| E_INHERIT Parent_list
Parent_list: -- /* empty */
| Parent
| Parent_list Parent
| Parent_list ';' Parent
```

```
Parent: Class_type Feature_adaptation_opt
Feature_adaptation_opt: -- /* empty */
| Feature_adaptation1
| Feature_adaptation2
| Feature_adaptation3
| Feature_adaptation4
| Feature_adaptation5
Feature_adaptation1: Rename New_exports_opt Undefine_opt Redefine_opt
Select_opt E_END
Feature_adaptation2: New_exports Undefine_opt Redefine_opt Select_opt E_END
Feature_adaptation3: Undefine Redefine_opt Select_opt E_END
Feature_adaptation4: Redefine Select_opt E_END
Feature_adaptation5: Select E_END
Rename: E_RENAME Rename_list
Rename_list: -- /* empty */
| Feature_name E_AS Feature_name
| Rename_list ',' Feature_name E_AS Feature_name
New_exports: E_EXPORT New_export_list
New_exports_opt: -- /* empty */
| New_exports
New_export_list: -- /* empty */
| New_export_item
| New_export_list New_export_item
| New_export_list ';' New_export_item
New_export_item: Clients Feature_set
Feature_set: Feature_list
| E_ALL
Feature_list: -- /* empty */
| Feature_name
| Feature_list ',' Feature_name
Clients: '{' Class_list '}'
Clients_opt: -- /* empty */
| Clients
Class_list: -- /* empty */
| E_IDENTIFIER
| Class_list ',' E_IDENTIFIER
Redefine: E_REDEFINE Feature_list
Redefine_opt: -- /* empty */
| Redefine
Undefine: E_UNDEFINE Feature_list
Undefine_opt: -- /* empty */
| Undefine
Select: E_SELECT Feature_list
Select_opt: -- /* empty */
| Select
Exheritance_opt: -- /* empty */
| E_EXHERIT Heir_list Exherited_feature_list Foster_adaptation_opt
Heir_list: -- /* empty */
| Heir
| Heir_list Heir
| Heir_list ';' Heir
Heir: Class_type Feature_adaptation_opt
Feature_adaptation_opt: -- /* empty */
| Feature_adaptation11
```

```
| Feature_adaptation12
| Feature_adaptation13
| Feature_adaptation14
Feature_adaptation11: Rename Adapt_opt Moveup_opt Select_RI_opt E_END
Feature_adaptation12: Adapt Moveup_opt Select_RI_opt E_END
Feature_adaptation13: Moveup Select_RI_opt E_END
Feature_adaptation14: Select_RI E_END
Adapt: E_ADAPT Feature_list
Adapt_opt: -- /* empty */
| Adapt
Moveup: E_MOVEUP Feature_list
Moveup_opt: -- /* empty */
| Moveup
Select_RI: E_SELECT Qualified_Feature_list
Select_RI_opt: -- /* empty */
| Select_RI
Qualified_feature_list: Feature_list_RI
| Qualified_feature_list ';' Feature_list_RI
Feature_list_RI: Feature_list
| Descendant_qualification ':' Feature_list
Descendant_qualification: Class_name
| Descendant_qualification '.' Class_name
Exherited_feature_list:
E_ONLY Feature_list | E_EXCEPT Feature_list | E_ALL | E_NOTHING
Foster_adaptation_opt:
New_exports_opt Redefine_opt
-- New_exports_opt from Inheritance
-- Redefine_opt from Inheritance
Creators_opt: -- /* empty */
| Creation_clause
| Creators_opt Creation_clause
Creation_clause: E_CREATION Clients_opt Procedure_list
Procedure_list: -- /* empty */
| E_IDENTIFIER
| Procedure_list ',' E_IDENTIFIER
Features_opt: -- /* empty */
| Feature_clause
| Features_opt Feature_clause
Feature_clause: E_FEATURE Clients_opt Feature_declaration_list
Feature_declaration_list: -- /* empty */
| Feature_declaration
| Feature_declaration_list Feature_declaration
| Feature_declaration_list ';' Feature_declaration
Feature_declaration: New_feature_list Declaration_body
Declaration_body: Formal_arguments_opt Type_mark_opt Constant_or_routine_opt
Adapted_opt
Constant_or_routine_opt: -- /* empty */
| E_IS Feature_value
Feature_value: Manifest_constant
| E_UNIQUE
| Routine
New_feature_list: New_feature
| New_feature_list ',' New_feature
New_feature: Feature_name
```

```
| E_FROZEN Feature_name
Feature_name: E_IDENTIFIER
| E_PREFIX E_STRING
| E_INFIX E_STRING
Formal_arguments_opt: -- /* empty */
| '(' Entity_declaration_list ')'
Entity_declaration_list: -- /* empty */
| Entity_declaration_group
| Entity_declaration_list Entity_declaration_group
| Entity_declaration_list ';' Entity_declaration_group
Entity_declaration_group: Identifier_list ':' Type
Identifier_list: E_IDENTIFIER
| Identifier_list ',' E_IDENTIFIER
Type_mark_opt: -- /* empty */
| ':' Type
Routine: Obsolete_opt Precondition_opt Local_declarations_opt
Routine_body Postcondition_opt Rescue_opt E_END
Routine_body: E_DEFERRED
| E_DO Compound
| E_ONCE Compound
| E_EXTERNAL E_STRING External_name_opt
External_name_opt: -- /* empty */
| E_ALIAS E_STRING
Local_declarations_opt: -- /* empty */
| E_LOCAL Entity_declaration_list
Precondition_opt: -- /* empty */
| E_REQUIRE Assertion
| E_REQUIRE E_ELSE Assertion
| E_REQUIRE E_OTHERWISE Assertion -- New for RI!
Postcondition_opt: -- /* empty */
| E_ENSURE Assertion
| E_ENSURE E_THEN Assertion
| E_ENSURE E_OTHERWISE Assertion -- New for RI!
Invariant_opt: -- /* empty */
| E_INVARIANT Assertion
Assertion: -- /* empty */
| Assertion_clause
| Assertion Assertion_clause
| Assertion ';' Assertion_clause
Assertion_clause: Expression
| E_IDENTIFIER ':' Expression
Adapted_opt: /* empty */
| E_ADAPTED Adapted_list E_END
Adapted_list: Adapted_item
| Adapted_list Adapted_item
| Adapted_list ';' Adapted_item
Adapted_item: '{' Class_type_list '}' Attribute_adaptation
| '{' Class_type_list '}' Routine_adaptation
Class_type_list: Class_type
| Class_type_list ',' Class_type
Attribute_adaptation:
Adapted_type E_IS '(' Expression ')' Adapted_result
Adapted_type: Type
| E_LIKE E_PRECURSOR
```

```
Adapted_result: ':' Expression
-- May contain 'Result'.
Routine_adaptation:
Adapted_formals Adapted_type_mark_opt E_IS
Adapted_actuals Adapted_result_opt
| Adapted_type E_IS Adapted_result
Adapted_formals: '(' Entity_declaration_list ')'
| '(' E_LIKE E_PRECURSOR ')'
Adapted_type_mark_opt: Type_mark_opt
| ':' E_LIKE E_PRECURSOR
Adapted_actuals: '(' Actual_list ')'
| '(' E_PRECURSOR ')'
-- The expressions in Actual_list may contain names of formal arguments
-- of the foster class routine.
Adapted_result_opt: /* empty */
| Adapted_result
Rescue_opt: -- /* empty */
| E_RESCUE Compound
Type: Class_type
| E_EXPANDED Class_type
| E_SEPARATE Class_type
| E_LIKE E_CURRENT
| E_LIKE E_IDENTIFIER
| E_BITTYPE Integer_constant
| E_BITTYPE E_IDENTIFIER
Class_type: E_IDENTIFIER Actual_generics_opt
Actual_generics_opt: -- /* empty */
| '[' Type_list ']'
Type_list: -- /* empty */
| Type
| Type_list ',' Type
Compound: -- /* empty */
| Instruction
| Compound Instruction
Instruction: Creation
| Call
| Assignment
| Conditional
| Multi_branch
| Loop
| Debug
| Check
| E_RETRY
| ';'
Creation: '!' Type '!' Writable Creation_call_opt
| E_BANGBANG Writable Creation_call_opt
Creation_call_opt: -- /* empty */
| '.' E_IDENTIFIER Actuals_opt
Assignment: Writable Assign_op Expression
Assign_op: E_ASSIGN
| E_REVERSE
Conditional: E_IF Expression E_THEN Compound Elseif_list Else_part E_END
Else_part: -- /* empty */
| E_ELSE Compound
```

```
Elseif_list: -- /* empty */
| E_ELSEIF Expression E_THEN Compound
| Elseif_list E_ELSEIF Expression E_THEN Compound
Multi_branch: E_INSPECT Expression When_list Else_part E_END
When_list: -- /* empty */
| E_WHEN Choices E_THEN Compound
| When_list E_WHEN Choices E_THEN Compound
Choices: -- /* empty */
| Choice
| Choices ',' Choice
Choice: Choice_constant
| Choice_constant E_DOTDOT Choice_constant
Choice_constant: E_IDENTIFIER
| Integer_constant
| E_CHARACTER
Loop: E_FROM Compound Invariant_opt Variant_opt E_UNTIL Expression
E_LOOP Compound E_END
Variant_opt: -- /* empty */
| E_VARIANT -- Not standard.
| E_VARIANT Expression
| E_VARIANT E_IDENTIFIER ':' Expression
Debug: E_DEBUG Debug_keys_opt Compound E_END
Debug_keys_opt: -- /* empty */
| '(' Debug_key_list ')'
Debug_key_list: -- /* empty */
| E_STRING
| Debug_key_list ',' E_STRING
Check: E_CHECK Assertion E_END
Call: Call_chain
| E_RESULT '.' Call_chain
| E_CURRENT '.' Call_chain
| '(' Expression ')' '.' Call_chain
| E_PRECURSOR Actuals_opt
| E_PRECURSOR Actuals_opt '.' Call_chain
| '{' E_IDENTIFIER '}' E_PRECURSOR Actuals_opt
| '{' E_IDENTIFIER '}' E_PRECURSOR Actuals_opt '.' Call_chain
Call_chain: E_IDENTIFIER Actuals_opt
| Call_chain '.' E_IDENTIFIER Actuals_opt
Actuals_opt: -- /* empty */
| '(' Actual_list ')'
Actual_list: -- /* empty */
| Actual
| Actual_list ',' Actual
Actual: Expression
| '$' Address_mark
Address_mark: Feature_name
| E_CURRENT
| E_RESULT
Writable: E_IDENTIFIER
| E_RESULT
Expression: Call
| E_RESULT
| E_CURRENT
| E_PRECURSOR
```

```
| '(' Expression ')'
| Boolean_constant
| E_CHARACTER
| E_INTEGER
| E_REAL
| E_STRING
| E_BIT
| E_LARRAY Expression_list E_RARRAY
| '+' Expression %prec E_NOT
| '-' Expression %prec E_NOT
| E_NOT Expression
| E_FREEOP Expression %prec E_NOT
| Expression E_FREEOP Expression
| Expression '+' Expression
| Expression '-' Expression
| Expression '*' Expression
| Expression '/' Expression
| Expression '^' Expression
| Expression E_DIV Expression
| Expression E_MOD Expression
| Expression '=' Expression
| Expression E_NE Expression
| Expression '<' Expression
| Expression '>' Expression
| Expression E_LE Expression
| Expression E_GE Expression
| Expression E_AND Expression
| Expression E_OR Expression
| Expression E_XOR Expression
| Expression E_AND E_THEN Expression %prec E_AND
| Expression E_OR E_ELSE Expression %prec E_OR
| Expression E_IMPLIES Expression
| E_OLD Expression
| E_STRIP '(' Attribute_list ')'
Attribute_list: -- /* empty */
| E_IDENTIFIER
| Attribute_list ',' E_IDENTIFIER
Expression_list: -- /* empty */
| Expression
| Expression_list ',' Expression
Manifest_constant: Boolean_constant
| E_CHARACTER
| Integer_constant
| Real_constant
| E_STRING
| E_BIT
Boolean_constant: E_TRUE
| E_FALSE
Integer_constant: E_INTEGER
| '-' E_INTEGER
| '+' E_INTEGER
Real_constant: E_REAL
| '-' E_REAL
| '+' E_REAL
```

List of Algorithms

List of Figures

List of Tables

Bibliography

[AG00] K. Arnold and J. Gosling. *The Java Programming Language*. Sun Microsystems, 3rd edition, USA, 2000.

[Bez07] Eric Bezault. GOBO Eiffel Project. http://www.gobosoft.com, November 2007.

[BI08] Borland International. Borland C++ 5.5 - Command-line compiler, 2008.

[Chi06] Ciprian-Bogdan Chirila. State of the art in reuse mechanisms of object-oriented programming. Phd Research Report #1, University Politehnica Timisoara, Romania, March 2006.

[Chi08] Ciprian-Bogdan Chirila. The model of the generic mechanism for the extension of object-oriented programming languages reverse inheritance for eiffel. Phd Research Report #2, University Politehnica Timisoara, Romania, February 2008.

[Cor08a] Microsoft Corporation. Microsoft Windows, 2008.

[Cor08b] Microsoft Corporation. Visual C++, 2008.

[FOP04] Oleksandr Fuks, Jonathan S. Ostroff, and Richard F. Paige. SECG: The SCOOP-to-Eiffel code generator. 3:143–160, 2004.

[Fou95] GNU Software Foundation. Flex - a fast scanner generator. http://www.gnu.org/software/flex, March 1995.

[Fou06] GNU Software Foundation. Bison - GNU parser generator. http://www.gnu.org/software/bison, 2006.

[GHJV97] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1997.

[Joh79] Steven C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979.

[KK02] Günter Kniesel and Helge Koch. ConTraCT - conditional transformations for incremental compilation of aspects. Demonstration at 1st International Conference on Aspect-Oriented Software Development, June 2002.

[Kni06] Günter Kniesel. A logic foundation for conditional program transformations. Technical Report IAI-TR-2006-1, Computer Science Department III, University of Bonn, January 2006.

[SF08] GNU Software Fundation. GCC, the GNU compiler collection, 2008.

[Sof08] Eiffel Software. Eiffel studio, 2008.

[TOG08] The Open Group. The Unix System, 2008.