Electronic Communications of the EASST
Volume 3 (2006)

Proceedings of the Third Workshop on Software Evolution through Transformations: Embracing the Change (SeTra 2006)

From C++ Refactorings to Graph Transformations

László Vidács and Martin Gogolla and Rudolf Ferenc

15 pages

Guest Editors: Jean-Marie Favre, Reiko Heckel, Tom Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

From C++ Refactorings to Graph Transformations

László Vidács[1] and Martin Gogolla[2] and Rudolf Ferenc[1]

[1]Department of Software Engineering, University of Szeged, Hungary
[2]Department for Mathematics and Computer Science, University of Bremen, Germany

Abstract: In this paper, we study a metamodel for the C++ programming language. We work out refactorings on the C++ metamodel and present the essentials as graph transformations. The refactorings are demonstrated in terms of the C++ source code and the C++ target code as well. Graph transformations allow to capture refactoring details on a conceptual and easy to understand, but also very precise level. Using this approach we managed to formalize two major aspects of refactorings: the structural changes and the preconditions.

Keywords: Metamodel, C++, UML, Graph Transformation, OCL, Refactoring

1 Introduction

The programming language C++ is widely used in industry today. Many applications written in C++ exist which are constantly developed further, for example, to be adapted to modern service-oriented aspects. On the other hand, there is an important current trend in software engineering that focuses on development activities for using models instead of concentrating on code production only.

This contribution tries to narrow the bridge between industrial, code-centric development with C++ and model-centric development employing languages like the Unified Modeling Language (UML). We discuss a C++ metamodel and display first ideas how development and maintenance of C++ artifacts can be performed on instantiations of this C++ metamodel based on refactorings. Our proposal is to express C++ refactorings and development steps as graph transformations. We think it is important to clearly express transformation concepts for an involved domain like C++ software development. Graph transformations possess a sound theoretical basis and allow to express properties on a conceptual level, not only on an implementation level. Surprisingly, graph transformations have not yet been applied for C++ software development.

Graph transformations have been applied for the transformation of metamodels, see for example the work of Gogolla [Gog00] (among many other works on graph transformation on metamodels). In industry, refactoring techniques [Ref06b] are regarded as promising means for software development. Refactoring of C++ code is supported by a variety of tools [Sli06][Ref06a] [Xre06]. However, refactorings are usually considered from the implementation point of view only, not from a conceptual view. A conceptual view on C++ refactorings on the basis of metamodels and graph transformations allows to express properties like refactoring applicability more precise. A conceptual view also opens the possibility for viewing refactorings on the semantical level, for example, in order to describe semantics preserving refactorings or to test whether they are semantics preserving.

The paper is organized as follows. The next section introduces the C++ language metamodel used in this work. Section 3 discusses C++ refactorings on this metamodel in terms of graph

transformations. Section 4 gives insight to our implementation. In Section 5 we mention some important contributions of this area. Finally, the paper ends with a short conclusion.

2 C++ Metamodel

Metamodels, which are also called schemas in the re-engineering community, play a very important role in the process of source code analysis. They define the central repository for the whole process, from where the facts can be reached with the help of different transformations.

Several researchers have been working on defining metamodels for C++ programs for reverse engineering purposes (also for program comprehension or to define an exchange format) [EKRW02], [Bel00], [FSH+01].

The Columbus Schema for C++ [FBTG02] satisfies some important requirements of an exchange format. It reflects the low-level structure of the code, as well as higher level semantic information (e.g., semantics of types). Furthermore, the structure of the metamodel and the used standard notation (UML Class Diagrams) make its implementation straightforward, and what is even more important, an API (Application Programming Interface) is very simple to be realized as well.

Because of the high complexity of the C++ language, the metamodel is divided into six packages. To introduce all packages is beyond the scope of this paper. To clearly present our ideas we created an excerpt mainly from the *struc* package of the metamodel. The presented approach is not limited to this subset of the C++ language, for example templates are also supported by the metamodel. To carry out refactorings on a C++ program it is necessary to deal with preprocessor directives. Although we have a separate metamodel for the preprocessor directives [VBR04], coping with them is not included to this contribution. The excerpt of the C++ metamodel can be seen in Figure 1 The upper part in the figure represents the scoping structure of a C++ program. Class *Member* is the parent of all kinds of elements which may appear in a scope (we use the term “member” in a more general way than usual). In the excerpt there are three important subclasses of *Member*. Class *Class* stands for C++ classes. It may contain further members; it may have base classes (shown by *BaseSpecifier*) and friends. In C++ a friend (class or function shown by *FriendSpecifier* in the figure) can access also protected and private members of the class. The class *Function* stands for C++ functions. It has body (represented by *Block*) which contains any number of *Statement*s; and has parameters (class *Parameter*). The class *Object* represents both variables and member fields in a *Class*. In the lower left corner there are the necessary enumerations.

In the middle there are classes for type representation. Like in C, in C++ types can be complex, so each language element which has a type contains a wrapper class called *TypeRep*. A *TypeRep* contains *TypeFormers* (in complex cases a type consists of many code pieces, each piece is a type-former). In the figure many type-formers are omitted, only one is shown (*TypeFormerType*) which refers directly to a type (to a *Class* in this case). This typing structure enables to express all kinds of types and helps to avoid redundancies in storing types. In the lower right corner there are some expressions which are used in this paper (*FunctionCall*, *MemberSelection*, *Id*). An *Id* expression is an identifier in the code which refers to *Member* - in the metamodel this means that it may refer to both classes and class members.

These are the main classes used in our example. For further details please see [Fro06]. As

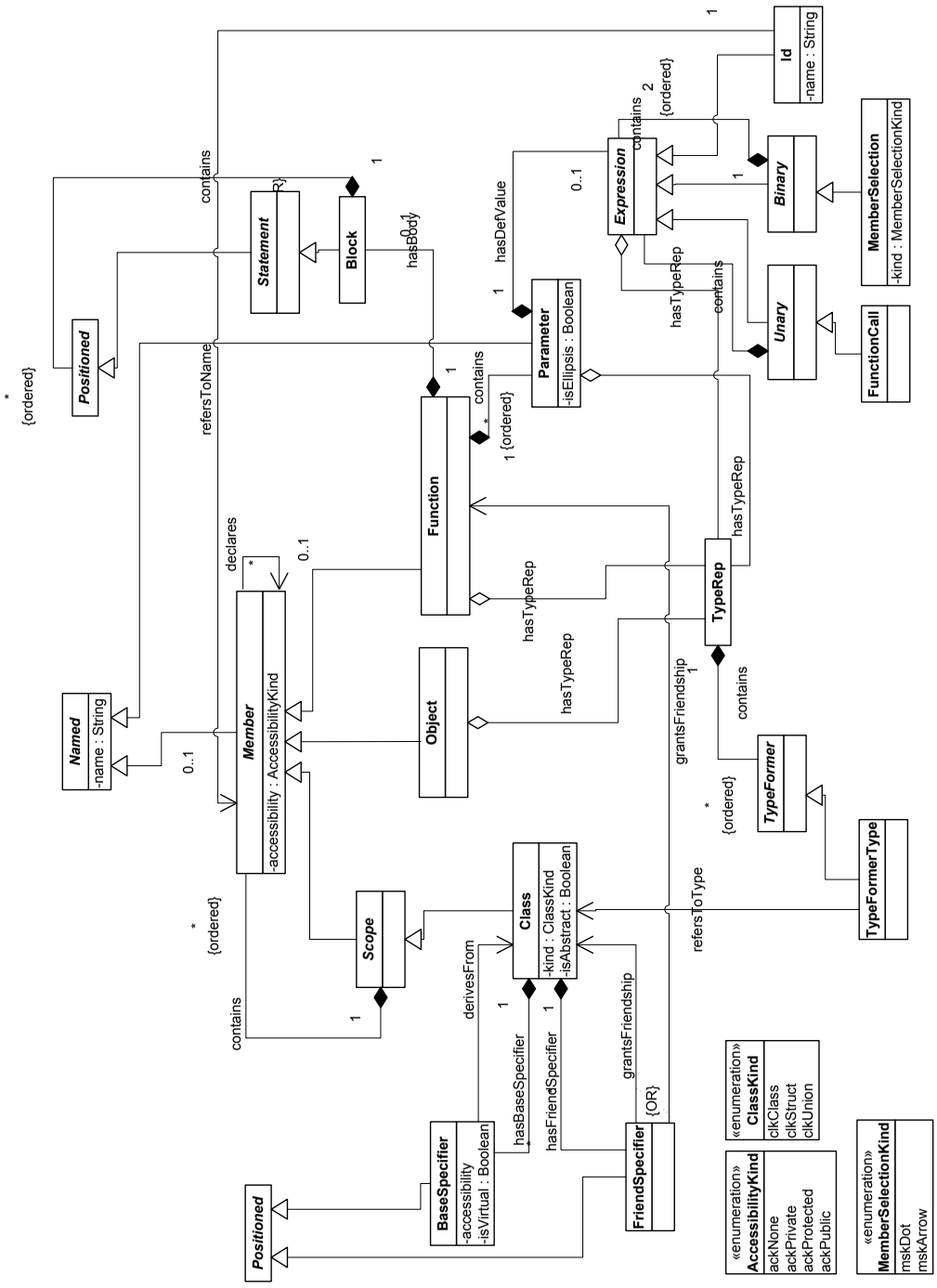

Figure 1: Excerpt from the Columbus Schema for C++ - UML class diagram

usual in case of complex class diagrams, we cannot express everything using UML class diagram notations easily. For example a function body (block statement) contains ordered *Positioned* nodes. However a *Parameter* is also *Positioned* but it cannot be a part of a function body. OCL expressions as constraints of the class diagram solve many similar issues. We define the following condition (boolean expression, called invariant) that must be true for all *Block* objects:

```
context Block inv:
not self.contains.oclIsTypeOf(struc_Parameter)
```

In the following sections we use the introduced metamodel and the OCL expressions together.

3 Graph Transformation Rules on the C++ Metamodel

In this section we show how refactorings on the C++ metamodel can be described with graph transformation. We provide example, which are "classical" refactorings from Fowler's catalog [Ref06b]. The basic idea is simple in both cases, however many subtle details arise when realizing these refactorings. We also concentrate on C++-specific issues.

3.1 Graph transformation approach, notation

We use a single pushout approach for graph transformation rules possessing a left and a right hand side. Instead of too complicated NACs (Negative Application Condition) we provide preconditions as OCL expressions.

The definition of directed, attributed graphs are used as usual. A program graph is directed, labelled, attributed graph where:

- nodes are labelled with class names shown in the UML class diagram
- nodes have attributes which are called as class attributes, the possible values of the attributes are from the corresponding UML types
- edges are labelled with relation names shown in the UML class diagram

Program graphs are introduced using an object diagram-like notation (see Figure 2).

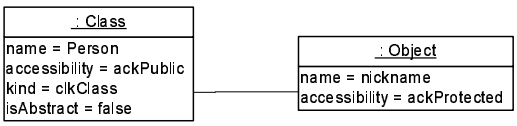

Figure 2: Object diagram like notation of the graph

Not all graphs that correspond to the definition above represent C++ programs. For example an undeclared variable may occur according to the metamodel but the belonging code could not compile. The well formedness is not checked in this paper. In reverse engineering context we assume that the starting graph is a well-formed graph and this property is preserved due to the conditions of transformations. Note that C++ class attributes are modeled with class Object in

the metamodel - so the nodes representing them have label *Object*. All *Object* and *Function* nodes have *TypeRep* nodes which show their C++ type. These nodes are omitted from the figures and presented only where it aids comprehension. Furthermore the notation of multi-nodes is introduced to describe general subgraphs. A multi-node with value k represents k pieces of nodes of the same type. Usage of multi-nodes is shown in Figure 3.

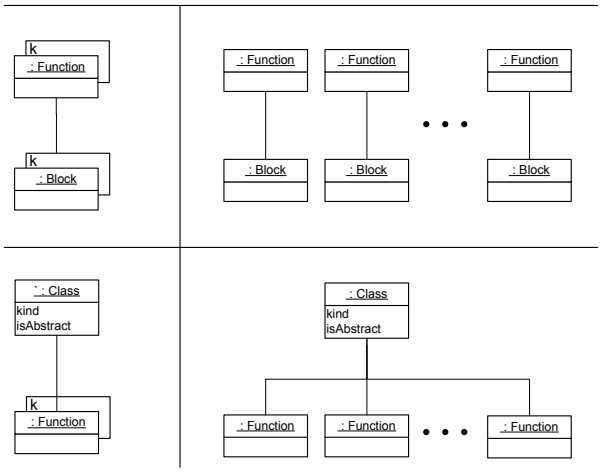

Figure 3: Usage of multi-nodes

3.2 Extract class

Classes should serve a clear, well-defined aim. During development, classes are growing. In lots of cases, there are new responsibilities added to them. The aim of this refactoring is to extract a separate concept and corresponding data to a new class to improve the quality of the design. The idea is shown in Figure 4 which is taken from the Refactoring catalog [Ref06b].

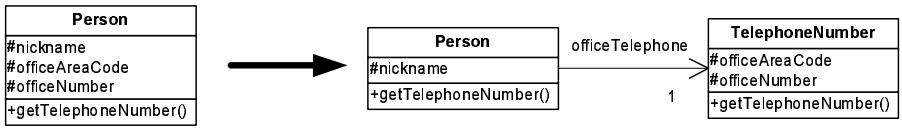

Figure 4: Extract class refactoring example

There is a long way from this semi-informal description to an applicable transformation. Based on this figure we make decisions about the context and purpose of the refactoring. After that we formalize it as graph transformation in two steps: the first part concentrates on the structure of the rule and the second part declares conditions using OCL expressions.

The main questions to be considered when realizing this refactoring as a graph transformation is: who can use the new (extracted) class and how can it be used. At first, the old class must somehow access it. If it is the only class that uses the new class then their relationship can be implemented either as the new class is a member in the old class or as a dynamic object creation

in the constructor of the old class and as a deletion of the object in the destructor of the old class (in this case the other classes can access the new class through public interface functions of the old class). On the other hand, if the new class is free to be used by other classes then it is more complicated - for example a reference counter can be used. This is related to the question: who can instantiate the new class. Similarly, it has to be determined how the new class can be accessed by other classes: through the old class only or through public functions of the new class as well. Another obstacle is introduced by attributes which are used externally from the old class and are now moved to the new class. We assume that the visibility enables one to access those members through a known interface only (this is not a constraint, using the C++ metamodel, all usages of an attribute can be checked).

In this paper we choose to protect the newly created class. The new class can be instantiated only by the old class. Its properties can be modified by the old class, furthermore the old class provides the public interface to use the new class.

As refactorings can be realized in many different ways so we formalize the extract class first as a rule schema. A rule schema has parameters and multi nodes and can be instantiated in concrete cases. We call rule only these concrete cases when the rule schema has concrete arguments and can be applied directly on a program graph. The attributes and operations to be moved must be determined by analyzing their usage. Therefore these are parameters of the transformation: any number of `Object` and `Function` nodes which are contained by the old class. The graph transformation rule schema of ExtractClass(... : (Object—Function)) is shown in Figure 5.

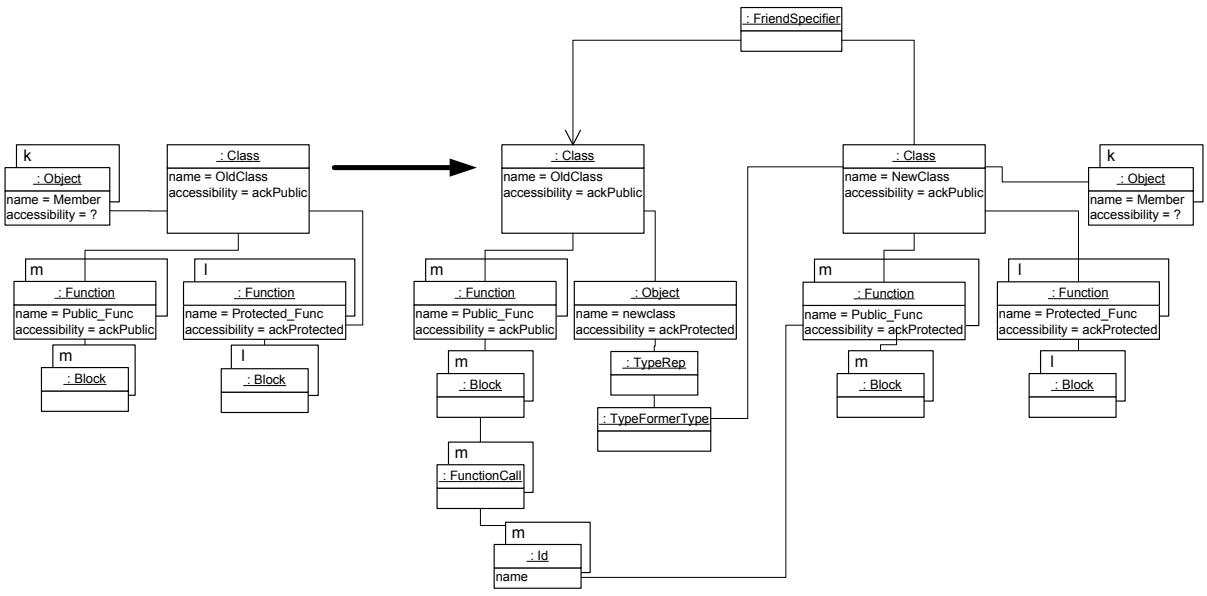

Figure 5: Extract class refactoring as graph transformation

On the left hand side there is the old class. Its members are divided into 3 groups: attributes, public (interface) functions and protected functions. On the right hand side there are both the old class and the new (extracted) class. The selected attributes and protected functions are com-

pletely moved to the new class. Public functions are copied from the old class to the new class. The existing implementation of these functions goes to the copy in the new class. The remaining functions in the old class have a new implementation: they only have to call the copied functions in the new class. The new class cannot be accessed from outside, so the copied public functions became protected.

To ensure the connection between the two classes we have a new member (*Object*) in the old class, its type is the new class. This is represented by new *TypeRep* and *TypeFormerType* nodes. (This abstraction of types is required to represent complex types in C++ [FBTG02].) Now we have to let the old class access the new one, which has protected members/functions. This is done by giving friendship grant to the old class. This is represented by the *FriendSpecifier* node.

Concrete rule and OCL conditions

The left hand side and right hand side of a concrete rule is given in the figures below. The C++ code before the transformation with the left hand side is shown in Figure 6. The graph is in fact the object model of the code, so it contains a node (nickname) which does not take part in the transformation. The resulting source code and the right hand side is shown in Figure 7.

```
class Person {
protected:
  string nickname;
  string officeAreaCode;
  string officeNumber;
public:
  string getTelephoneNumber();
};
```

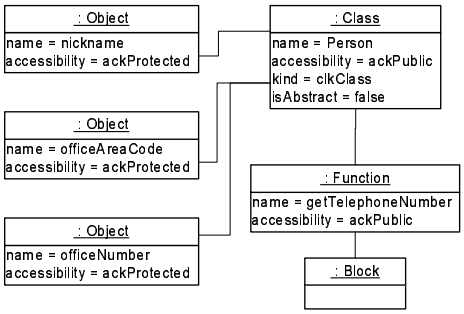

Figure 6: Extract class - C++ code and model instance before the transformation

```
class Person {
protected:
  string nickname;
  TelephoneNumber _TelephoneNumber;
public:
  string getTelephoneNumber();
};
class TelephoneNumber {
protected:
  TelephoneNumber();
  string officeAreaCode;
  string officeNumber;
  string getTelephoneNumber();
  friend class Person;
};
```

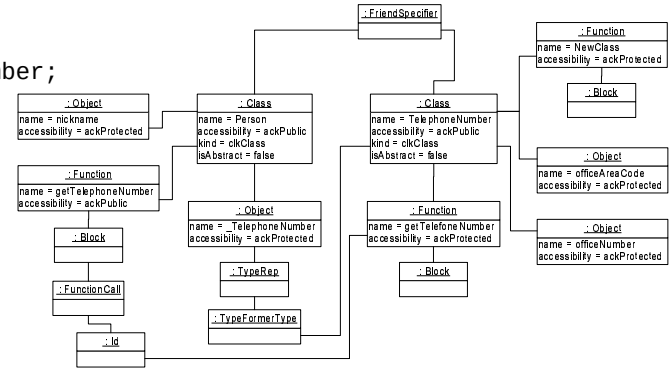

Figure 7: Extract class - transformed C++ code and model instance

Moving function from one class to another requires careful examinations of the body. Class members or functions referenced from the body may become unaccessible because of changing the class (they are ”foreign” in the new class). (Note that the friendship relation is not symmetric, only the old class can reach the new class.) To prevent accessing unreachable class members we have to check the subgraph of the function body. References to class members/functions are classified based on the relation of the old class and the referenced class as follows: inside the class, class hierarchy (base classes upwards) and outer classes. Bad references that prevent applying the rules are the following ones:

- reference to protected/private member of the old class which is not among the parameters of the rule (if the referenced member is public it means that the extract class refactoring is not so reasonable here)
- reference to protected/private member of one of the base classes (in public case the above note applies)
- reference to protected/private member of an outer class which is a friend of the old class

According to the metamodel a reference is an Id node which has a refersToName relation to a Member, especially to a Function or Object. The referenced Member is contained by a class which is the class we are looking for. To distinguish the 3 different cases mentioned above it is not enough to search Ids in the subgraph in special container nodes like MemberSelection or FunctionCall. It may happen that a member selection contains this pointer and the function call without a memberselection may reference a function in a base class. So we have to scan all Ids and find the referenced class members/functions and their container classes. Note that in the case of MemberSelection expressions the container class can be found through the left hand side child Id as well.

The scan can be implemented as an OCL expression.

```
let Old : struc_Class = B1.hasBody.containsMember.oclAsType(struc_Class)
in
let Bases : Bag(struc_Class)  = Old.hasBaseSpecifier.derivesFrom
in
let M : Bag (struc_Member)  = B1.containsPositioned->select(i | i.oclIsTypeOf(expr_Id)).
        oclAsType(expr_Id).refersToMember.select(oclIsTypeOf(struc_Object) or
        oclIsTypeOf(struc_Function))
in
M.iterate( m : struc_Member; res : Boolean = true |
let Cont : struc_Class = m.contains.oclAsType(struc_Class)
in
--condition A : referenced members are in the old class but they are not
--              among the parameters of the transformation rule
if ((m.accessibility='protected') or (m.accessibility='private') and
    (Cont=Old) and not (Bag{O2,O3,F1}.exists(i | i=m)) )
then
res and false
else
--condition B : referenced members are in the base classes
  if ((m.accessibility='protected') or (m.accessibility='private') and
```

```
      (Bases.exists(bc | bc=Old)))
  then
  res and false
  else
    --condition C : referenced members are outer friends
    let Friends : Bag(struc_Class) = Old.hasFriendSpecifier.grantsFriendship
    in
    if ((m.accessibility='protected') or (m.accessibility='private') and
        (Friends.exists(fc | fc=Cont)))
    then
    res and false
    else
    res
    endif
  endif
endif
```

The expression checks for wrong references in a function body B1 (argument of the expression). In case of any occurrence of a wrong reference the expression returns false and prevents the rule to be applied. This example show the expressiveness of the OCL. OCL expressions may contain searches through collections which cannot be easily formulated with simple NACs. A general subgraph notation is needed for instance to check whether a class is one of the base classes of the old class.

3.3 Other refactorings

There are several refactorings which can be implemented on our metamodel as graph transformations in a similar way. In this paper we give only the (detailed) case study of extract class not only because of space limitation. It contains many structural changes and also complex preconditions. In a work of Eetvelde [VJ05] there is a list of 15 formalized refactorings in 29 pages. Papers usually demonstrate two refactorings like pull up method and encapsulate variable [MVDJ05], or extract code and move method refactorings [BPT04]. We may say that presenting more of the above examples will not say more than our extract class example regarding the two important aspects of refactorings: formalization of the structural changes and the preconditions; which was the aim of the current contribution.

4 Implementation

The extract class refactoring graph transformation was implemented using the USE (UML-based Specification Environment) software [USE05] instead of an existing graph transformation engine. USE is a system for the specification of information systems. It is based on a subset of the Unified Modeling Language (UML). A USE specification contains a textual description of a model using features found in UML class diagrams (classes, associations, etc.). Expressions written in the Object Constraint Language (OCL) are used to specify additional integrity constraints on the model. A model can be animated to validate the specification against non-formal requirements.

The USE specification of C++ metamodel contains enumerations, classes and relations corresponding to the UML class diagrams of the metamodel. Every C++ program can be instantiated as an object diagram (a graph based on the specification). USE can handle and display our metamodel and model instances before and after the transformation fairly well. The transformation itself can not be handled directly in the environment. The left hand side and the right hand side of the transformation is modelled in the environment, both are saved to a text file. OCL expressions (used as postconditions to modify attributes of nodes) are added to this description. The description is processed by a script[1] which creates a sequence of basic graph operations from it (create/delete nodes, insert/delete edges). After the USE command file of the rule is generated this way, the rule can be applied on any model in the USE environment. The script (based on the description) also generates a function that can list the possible nodes on which the rule can be applied and a function which applies the transformation (see Appendix A). There is a generated class (called RuleCollection) in the USE model which contains information and functions regarding to transformation rules. To apply the rule, one has to pass the appropriate nodes as parameters.

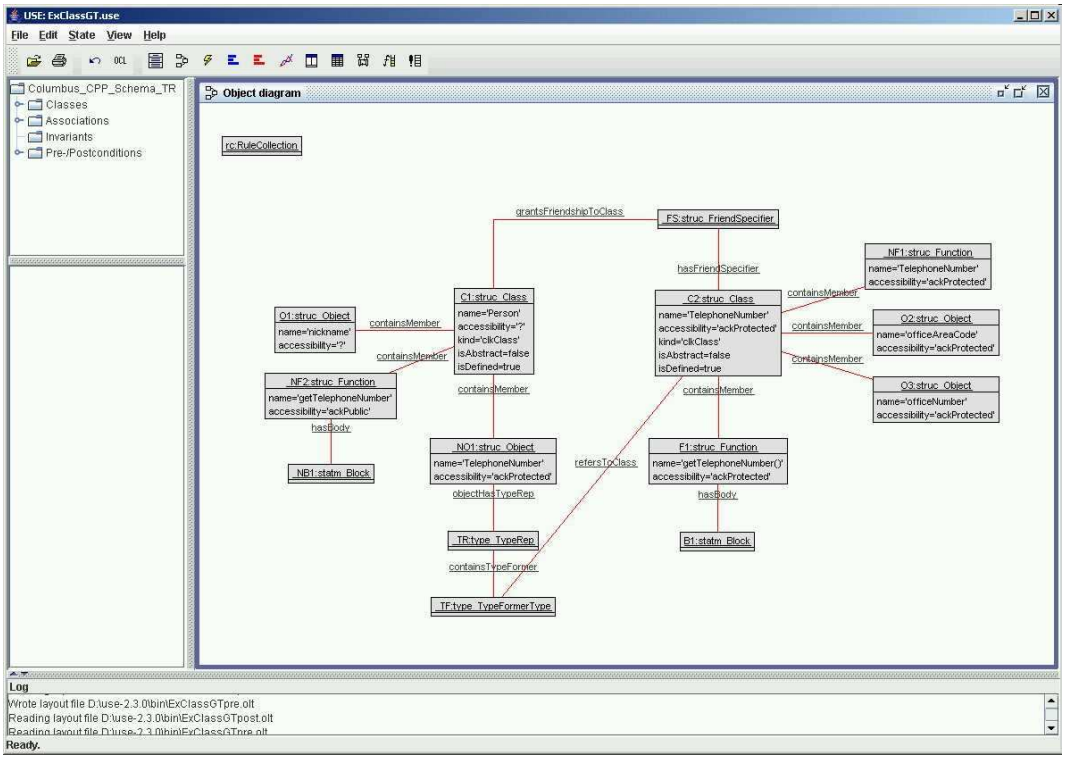

Figure 8: Object diagram in USE after the refactoring

The result of the extract class refactoring in USE can be seen in Figure 8. The execution of the script was quick because the parameters determined the place of the transformation so the modifications were made locally (below 1 sec). Future work is to try this implementation

[1] Thanks to Fabian Büttner

on real life software systems. Running time of the rule-creator script (which creates basic graph operations from a rule) depends on the size of the rule. In general the most time consuming part is identification of the places where the transformation is applicable. In case of refactorings in most cases the programmer has to consider and choose a place to apply the refactoring. The decision is made based on criteria which are not easy to formalize. For instance extract class refactoring may be applicable on almost every class (which has members or functions) but only in few cases it is useful to apply. Thus in a real life software the identification of the big class can be done using other visualization or analyzer tools. After we have identified the place (parameters of the rule), the actual refactoring can be applied quickly - like in our example.

5 Related work

Since the pioneering work of Opdyke [Opd92] there were lots of efforts made to give a formalism for refactorings. Graph transformations are also considered as a basis of formalizing refactorings. Our work has close connections to such approaches. A solid contribution that shows the current state of the art is given in the work of Mens at al. [MVDJ05]. The graph representation of a program plays an essential role in the formalism. The paper describes a language independent formalism and also introduces two major issues in detail: preconditions and behaviour preservation. We borrowed ideas of the graph formalization from Eetvelde at al. [VJ05] like the multi-nodes and edges. Bottoni [BPT04] uses a similar formalism, the focus in that work is on the coordination of a change in different model views of the code using distributed graph transformations. These works however are not specialized towards C++ as our approach is.

Although there is much progress in this area, industry uses more or less the same solutions as before: language specific refactorings are implemented separately. Fanta and Rajlich [FR98] contribute a natural way of implementing refactorings. The paper shows the key points but this solution is somehow “out of control” without a formal base. They state that these transformations are surprisingly complex and hard to implement. Two reasons they give for that are the nature of object-oriented principles and the language specific issues. We agree with this view, our work shows how to deal with C++ specific issues on meta level with the checking possibilities provided by the OCL. Our work also differs from the others in that instead of the usual graph transformation engines we used a script based OCL solution of the USE system.

6 Conclusion

This paper presents work in progress on using graph transformations to express C++ refactorings in a clear way and on a conceptual level. Although graph transformations have been used for metamodel transformation of various languages, they have not been extensively used for C++. For a successful use of graph transformations, it is necessary to work on cumbersome subjects like C++ refactorings and the accompanying nasty details. We are aware of the fact that many of our concepts are already known from other successful application areas of graph transformation.

Future work will elaborate further C++ language refactorings. We will also improve these refactorings in a graph rewriting machine resp. modeling tool. We think that such an implementation will give insight into the application conditions and properties of C++ refactorings

on a conceptual level. Such an implementation will thus enable a deeper understanding of C++ refactorings.[2]

Bibliography

[Bel00] Bell Canada Inc. DATRIX – Abstract semantic graph reference manual. Montréal, Canada, version 1.2 edition, Jan. 2000.

[BPT04] P. Bottoni, F. Parisi-Presicce, G. Taentzer. Specifying Integrated Refactoring with Distributed Graph Transformations. *Lecture Notes in Computer Science*. 3062:220–235, 2004.

[EKRW02] J. Ebert, B. Kullbach, V. Riediger, A. Winter. GUPRO - Generic Understanding of Programs. *Electronic Notes in Theoretical Computer Science* 72(2), 2002.

[FBTG02] R. Ferenc, Á. Beszédes, M. Tarkiainen, T. Gyimóthy. Columbus - Reverse Engineering Tool and Schema for C++. In *ICSM 2002: Proceedings of the International Conference on Software Maintenance*. Pp. 172–181. IEEE Computer Society, Montreal, Canada, Oct. 2002.

[FR98] Richard Fanta and Vaclav Rajlich. Reengineering object-oriented code. In *ICSM 1998: Proceedings of the International Conference on Software Maintenance*, page 238, IEEE Computer Society. Washington, DC, USA, 1998.

[Fro06] Homepage of FrontEndART Ltd. `http://www.frontendart.com`, 2006.

[FSH+01] R. Ferenc, S. E. Sim, R. C. Holt, R. Koschke, T. Gyimóthy. Towards a Standard Schema for C/C++. In *WCRE 2001*. Pp. 49–58. IEEE Computer Society, Oct. 2001.

[Gog00] M. Gogolla. Graph Transformations on the UML Metamodel. In *GVMT'2000*. Pp. 359–371. Carleton Scientific, Waterloo, Ontario, Canada, 2000.

[MVDJ05] T. Mens, N. Van Eetvelde, S. Demeyer, D. Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 2005.

[Opd92] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA, 1992.

[Ref06a] Homepage of Ref++. `http://www.refpp.com`, 2006.

[Ref06b] Refactoring catalog. `http://www.refactoring.com/catalog/`, 2006.

[Sli06] Homepage of Slickedit. `http://www.slickedit.com/`, 2006.

[USE05] Homepage of USE. `http://www.db.informatik.uni-bremen.de/projects/USE/`, 2006.

[VBR04] L. Vidács, Á. Beszédes, F. Rudolf. Columbus Schema for C/C++ Preprocessing. In *CSMR 2004*. Pp. 75–84. IEEE Computer Society, Mar. 2004.

[VJ05] N. Van Eetvelde, D. Janssens. Refactorings as Graph Transformations. Technical report, University of Antwerp, February 2005. UA WIS/INF 2005/04.

[Xre06] Homepage of Xrefactory. `http://xref-tech.com`, 2006.

[2] László Vidács acknowledges the financial support provided through the European Community's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis.

Appendix

A USE description

```
--------------------------------------------------------------------
-- Extract Class refactoring
--------------------------------------------------------------------
rule ExClass
left
  C1:struc_Class
  O1:struc_Object
  O2:struc_Object
  F1:struc_Function
  (C1,O1): containsMember
  (C1,O2): containsMember
  (C1,F1): containsMember
right
  C1:struc_Class
  C2:struc_Class
  O1:struc_Object
  O2:struc_Object
  NO1:struc_Object
  F1:struc_Function
  NF1:struc_Function
  NF2:struc_Function
  NB1:statm_Block
  FS:struc_FriendSpecifier
  TR:type_TypeRep
  TF:type_TypeFormerType
  (C2,O1): containsMember
  (C2,O2): containsMember
  (C2,F1): containsMember
  (C2,NF1): containsMember
  (C1,NF2): containsMember
  (NF2,NB1): hasBody
  (C1,NO1): containsMember
  (NO1,TR): objectHasTypeRep
  (TR,TF): containsTypeFormer
  (TF,C2): refersToClass
  (C2,FS): hasFriendSpecifier
  (FS,C1): grantsFriendshipToClass
  -- postconditions
  [C2.name = 'TelephoneNumber']
  [C2.accessibility = 'ackProtected']
  [C2.kind = 'clkClass']
  [C2.isAbstract = false]
  [C2.isDefined = true]
  [NF1.name = 'TelephoneNumber']
  [NF1.accessibility = 'ackProtected']
  [F1.accessibility = 'ackProtected']
  [NO1.name = 'TelephoneNumber']
  [NO1.accessibility = 'ackProtected']
  [NF2.name = 'getTelephoneNumber']
  [NF2.accessibility = 'ackPublic']
end
```

```
-----------------------------------------------------------------------
-- Extract Class refactoring
-- ExClassGT_ExClass.cmd
-- ExClass(_C1,_O1,_O2,_F1)
-- the 'match' parameter can be bound with '!let match = ...'
-----------------------------------------------------------------------
!let _C1 = match->at(1)
!let _O1 = match->at(2)
!let _O2 = match->at(3)
!let _F1 = match->at(4)
!openter rc ExClass(_C1,_O1,_O2,_F1)

!create _C2 : struc_Class
!create _NO1 : struc_Object
!create _NF1 : struc_Function
!create _NF2 : struc_Function
!create _NB1 : statm_Block
!create _FS : struc_FriendSpecifier
!create _TR : type_TypeRep
!create _TF : type_TypeFormerType
!insert(_C2,_O1) into containsMember
!insert(_C2,_O2) into containsMember
!insert(_C2,_F1) into containsMember
!insert(_C2,_NF1) into containsMember
!insert(_C1,_NF2) into containsMember
!insert(_NF2,_NB1) into hasBody
!insert(_C1,_NO1) into containsMember
!insert(_NO1,_TR) into objectHasTypeRep
!insert(_TR,_TF) into containsTypeFormer
!insert(_TF,_C2) into refersToClass
!insert(_C2,_FS) into hasFriendSpecifier
!insert(_FS,_C1) into grantsFriendshipToClass
!set _C2.name := 'TelephoneNumber'
!set _C2.accessibility := 'ackProtected'
!set _C2.kind := 'clkClass'
!set _C2.isAbstract := false
!set _C2.isDefined := true
!set _NF1.name := 'TelephoneNumber'
!set _NF1.accessibility := 'ackProtected'
!set _F1.accessibility := 'ackProtected'
!set _NO1.name := 'TelephoneNumber'
!set _NO1.accessibility := 'ackProtected'
!set _NF2.name := 'getTelephoneNumber'
!set _NF2.accessibility := 'ackPublic'
!delete(_C1,_O1) from containsMember
!delete(_C1,_O2) from containsMember
!delete(_C1,_F1) from containsMember
!opexit
```

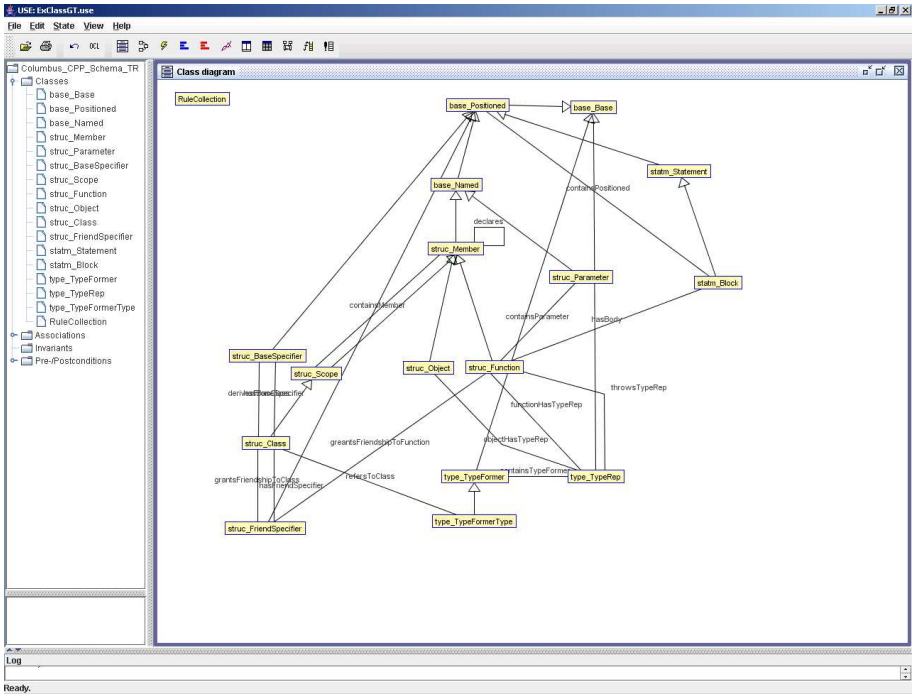

Figure 9: Class diagram of the C++ metamodel excerpt in USE

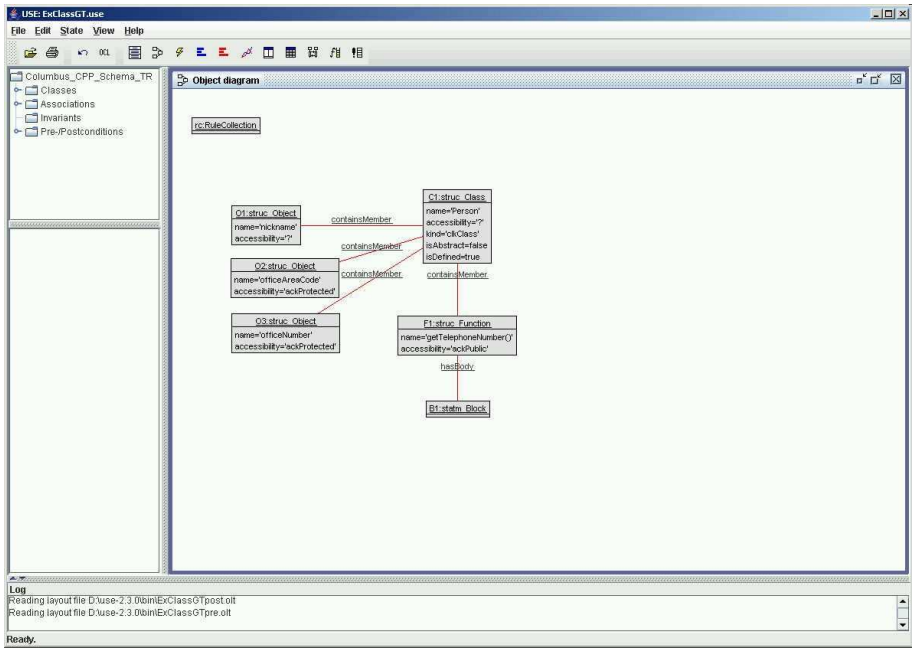

Figure 10: Object diagram in USE before the refactoring