REFACTORING OF C/C++ PREPROCESSOR CONSTRUCTS AT THE MODEL LEVEL

László Vidács
Research Group on Artificial Intelligence
University of Szeged and Hungarian Academy of Sciences, Szeged, Hungary

Keywords: Reverse engineering, Refactoring, Graph transformation, Preprocessor.

Abstract: Preprocessor directives are usually omitted from the analysis of C/C++ software, yet they play an important role especially in program transformations. Here a method is presented for refactoring preprocessor constructs at the model level. Refactorings are carried out on program models derived from a reverse engineering process of real-life software. We present a metamodel of preprocessing on which a graph transformation approach is used to elaborate refactorings. The method is presented through the elaboration of the add parameter refactoring both at schematic and concrete level. Safe transformations are assured by visual control and validated by the evaluation of OCL expressions. The usability of the idea is validated by successful experiments.

1 INTRODUCTION

In this paper we will carry out the refactoring of C/C++ preprocessor constructs at the model level. Our aim is to investigate transformations of reverse engineered program models derived from real-life C/C++ software, with the emphasis on the safety of the transformations.

In the past fifteen years refactoring has become an increasingly important technique for improving the design of existing code (Opdyke, 1992). It is a program transformation which preserves program behaviour, while the quality of the program becomes better from some point of view (reusability, maintainability, readability, flexibility). Nowadays refactoring is a well-known technique mainly due to the trend of model-driven development, including the emphasis on iterative development. Hence the strong need for tool support has resulted in an increasing number of refactoring tools, primarily for object-oriented languages like Smalltalk (Roberts et al., 1997), Java (jFactor, 2009), and for C++ (Slickedit Homepage, 2009); and for various kind of languages, many of them listed in the refactoring catalog (Refactoring Catalog, 2009). A refactoring by definition is a small transformation, but successively applied refactorings may lead to a larger modification. Therefore refactorings are usually applied together as composite refactorings. For example in an Add parameter refactoring, when a new parameter is added to a function, one has to consider modifying the call sites of the function.

Although refactoring in the C/C++ language is essential and affects many software developer companies, the tool support still has to be improved. One of the main reasons for this is that two languages actually have to be refactored: the C or C++ language itself, and the preprocessor language. The preprocessor has a textual syntax and works on tokens, while the C/C++ compiler uses a grammar with typical programming language constructs like variables and functions. Preprocessor constructs are usually neglected by C/C++ analyzer tools, but in the case of refactoring a tool is not usable unless it can handle them. In contrast to most studies, this current work uses the preprocessor as a separate language, so the preprocessor constructs are treated as the main subject of refactoring. Of course preprocessor refactorings may be used later in C/C++ language refactorings as well. There are many points which have to be considered before a refactoring can be applied (e.g. the so-called preconditions have to be fulfilled). The model-driven trend in software engineering allows one to carry out refactorings at the model level, including verifying the preconditions. In addition, the modified model may be further validated so that the concrete refactoring may be refined.

Our aim is to perform a sequence of refactorings on a reverse-engineered program model. The transformations on the preprocessor metamodel will be described by graph transformations, which make them easy to understand and handle (Gogolla, 2000). The *Columbus* and the *USE* systems were used to imple-

Vidács L. (2009).
REFACTORING OF C/C++ PREPROCESSOR CONSTRUCTS AT THE MODEL LEVEL.
In *Proceedings of the 4th International Conference on Software and Data Technologies*, pages 232-237
DOI: 10.5220/0002258302320237

ment our approach, which allows us to handle reverse-engineered program models and validate the transformations using *OCL* expressions. The paper is organized as follows: Section 2 contains our main contribution including a short description of the preprocessor metamodel, the graph transformation approach for refactoring, the reverse engineering process and the a case study on the add parameter refactoring. After, related works are discussed in Section 3, then in Section 4 we draw some pertinent conclusions.

2 REFACTORING ON THE C/C++ PREPROCESSOR METAMODEL

Refactoring is a way of improving the internal quality of the code (Mens and Tourwé, 2004), but internal quality is usually not among the top priorities of a typical software development team. Even when it is time to improve the maintainability or to correct the bad design of a former phase of the development, the estimated cost of a change may discourage refactoring activities. On the other hand, when refactoring is done on program code, testing after each refactoring step requires a great deal of effort, and in the worst cases major modifications may remain untested. Our intention here is to carry out controlled and validated refactoring steps (safe refactoring), then spend more time and effort on higher level or integration testing.

In this work refactorings were performed on program models produced by the Columbus Reverse Engineering Framework (Ferenc et al., 2002). *Columbus* builds a graph representation of C/C++ programs, which includes information about the use of preprocessor directives like macro definitions and calls. The graph representation conforms with a preprocessor metamodel. The model-level transformations are realized using the *USE* system (Gogolla et al., 2007). In this section graph transformation as a method for refactoring will be briefly described, followed by the necessary description of the preprocessor metamodel. In the remaining part the add parameter refactoring is investigated in detail. The scope of the current work covers the macro-related refactorings only.

2.1 Graph Transformations

The graph transformation approach is a natural way to express formal refactoring (e.g. refactoring at the model level). A graph transformation rule is usually given by two states of the graph (before and after the transformation), which corresponds to the usual view of a refactoring. Actually, there is a good correspondence between the notions and terms of refactoring and graph transformations (Mens and Tourwé, 2004). Mens et al. (Mens et al., 2005) give details on using graph transformations for refactoring and provide helpful illustrative examples. In addition to the general approach, efforts have been made towards applying this approach in domain specific environments as well (Taentzer et al., 2007).

Our current work began along similar lines as that presented in (Vidács et al., 2006). We use a single pushout approach for graph transformation rules possessing a left and a right hand side. Instead of using the rather complicated *NAC* (Negative Application Condition), we provide preconditions as *OCL* expressions. Despite the *NAC* concept being an integral part of the graph transformation theory, we will omit it because *OCL* is the natural way to express preconditions and it is also flexible enough to handle complex cases. The definitions of directed, attributed graphs are used in the usual way. A program graph is a directed, labelled, attributed graph where nodes, attributes and edges correspond to the metamodel we shall employ. Not all graphs that correspond to the definition above represent valid C/C++ preprocessor constructions, but in the reverse engineering context we shall assume that the starting graph is a well-formed graph and that this property is preserved because of the conditions of the transformations.

2.2 The Preprocessor Metamodel

Here we used the Columbus Schema for C/C++ Preprocessing as a preprocessor metamodel, which is represented in Figure 1 as an UML Class Diagram (Vidács et al., 2004). The structure of the metamodel follows the structure of a source file. From a preprocessor point of view, a file consists of elements which can be either preprocessor directives or other text elements. There are classes which describe both object-like and function-like (parameterized) macro definitions. Macro expansions can be tracked with the help of reference (*DefineRef*) objects. A *DefineRef* object links the position of the call (such as an *Id*) to the position of the macro definition. Also, function-like macro expansions contain an ordered list of arguments (*Argument* objects), which are matched to macro parameters (in this case the reference object is called *FuncDefineRef*). The structure of the macro body (replacement list) is described using the *DirectiveText* class and its descendant classes, with the help of associations between them. A macro definition may contain further macro calls, so that a sequence of expansions takes place during the full expansion of

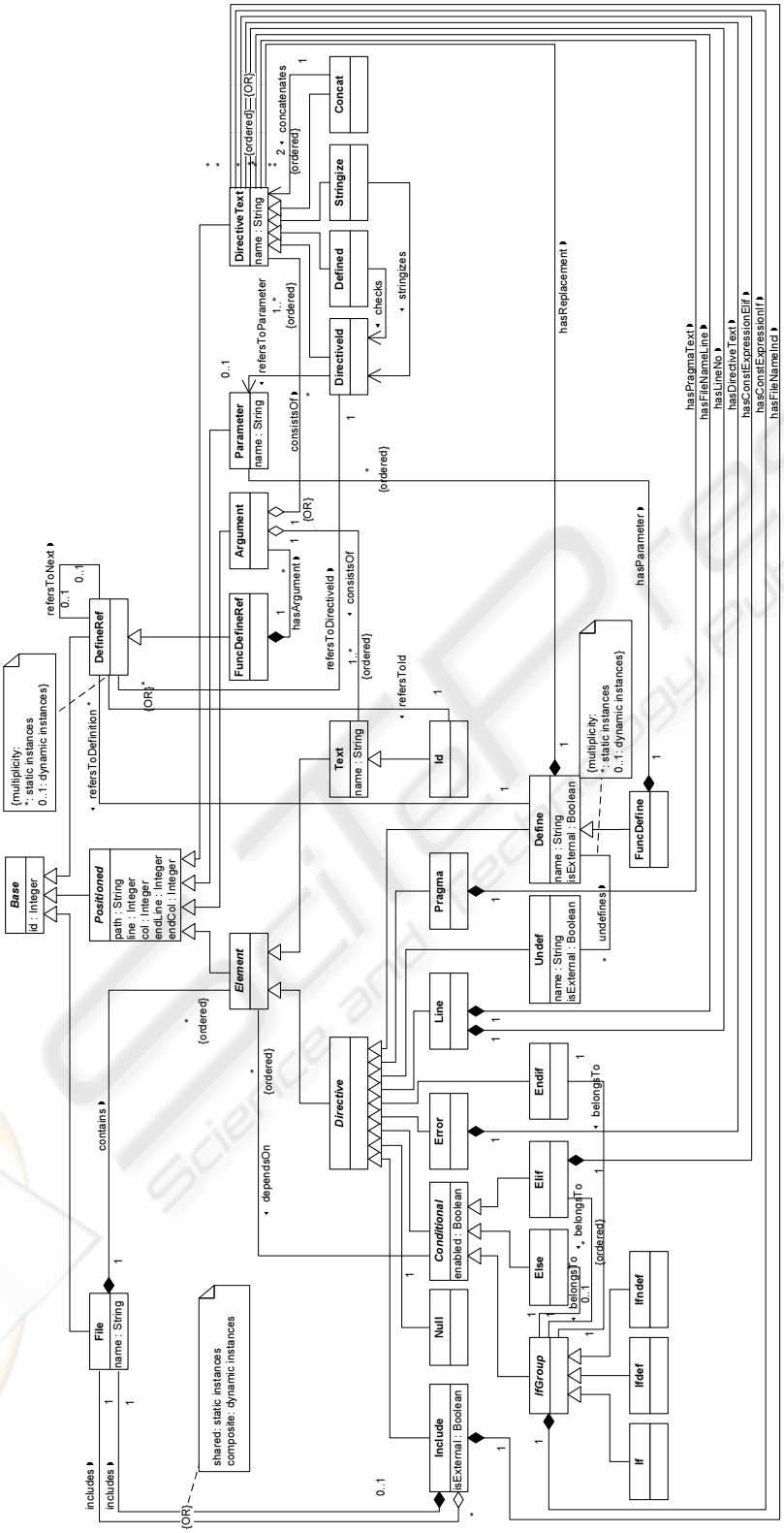

Figure 1: The preprocessor metamodel.

a macro. Since each expansion step requires a *DefineRef* object, a full expansion is represented by a sequence of reference objects, which is described by the *refersToNext* relation. For a detailed description of the schema see (Vidács et al., 2004).

2.3 Add Parameter Refactoring

The add parameter refactoring allows a method to process more information than it could previously. It may be a part of a complex refactoring, and it may implement a new feature as well. The object-oriented version of this refactoring (add parameter to a method) is described in (Fowler, 2002). In the object-oriented case there are several alternatives to consider such as whether to introduce a parameter object, or whether to get the required information through an object which has already been passed to the method. The proposed mechanics of changes in the code include the following steps: declare a new method with the additional parameter, copy the method body from the old one to the new one, compile the code, call the new method from the body of the old one, compile and test. Next, modify each call site of the old method to call the new one, compile and test it for each case, remove the old method, and finally compile and test.

In our case, we suggest the following steps:

- Check for alternatives, and avoid an excessively long parameter list
- Check for preconditions
- Determine the concrete type of the transformation rule and process
- Modify the call sites: add a new argument
- Check each call site by hand

Unlike code refactoring, the operations listed above are automated at the model level, except for the first and the last one.

Alternatives. In a rare case an alternative may be to get the required information from an existing parameter either by concatenation of an appropriate string or another parameter, or by using the stringize operator. One possibility is to get an enumerator name from an existing string parameter, or to get the string form of a case label. Some preprocessor implementations support variadic macros (variable parameter list), which in some cases make it unnecessary to add a new parameter. However a good reason for not making use of this refactoring is the long parameter list bad smell, which can be avoided by restructuring macros.

Preconditions. The applicability of the refactoring depends on the way the macros are defined. There are four types of macros based on the place of their definition:

- Standard macros - defined by the preprocessor standard, e.g. `__FILE__`, `__LINE__`
- Environment macros - defined by the compiler environment, e.g. `__GCC_VER__` for *GCC* or `MSC_VER` for *Microsoft Visual Studio*.
- Command line macros - defined as command line parameters, applied to the actual source file only.
- Ordinal macros in the source code

In the metamodel the type of a macro is represented by the *isExternal* attribute. The value is true in the case of standard macros, environment macros, and command line defined macros. Naturally, refactorings may only be applied to the non-external definitions.

The new parameter name has to fulfill some conditions. There should not exist a parameter with the same given name. The replacement text of the macro must be checked for the new parameter name to see whether a preprocessing token already exists with the same text. These conditions can be checked locally. (Note that there is no need to check whether the new name conflicts with an already defined macro name.)

Concrete Type of Transformation. A general refactoring in most cases can be formalized in many ways depending on the specific needs. Similarly, a graph transformation can be presented as a transformation rule schema and a set of concrete transformations. The add parameter refactoring includes three types of transformations: function-like, object-like and variadic macros. Due to space limitations only the first one is elaborated here. The schematic graph transformation rule for the add parameter to a function-like macro is presented in two figures. Figure 2 contains the left hand side of the transformation and Figure 3 contains the right hand side after the refactoring. The macro definition part is shown on the left hand side of both figures, while on the right hand side of each figure there is an example call of the macro. In Figure 3 the new nodes are shown in grey and the new edges are depicted in bold. The schematic program code corresponding to both sides of the transformation is the following for the left hand side:

```
#define MACRO(P1, ... Pn) \
          R1 ... R_P1 ... R_Pn ... Rm
...
MACRO (A1, ... An)
```

and for the right hand side it is:

```
#define MACRO(P1, ... Pn, Pn+1) \
          R1 ... R_P1 ... R_Pn ... Rm
...
MACRO (A1, ... An, An+1)
```

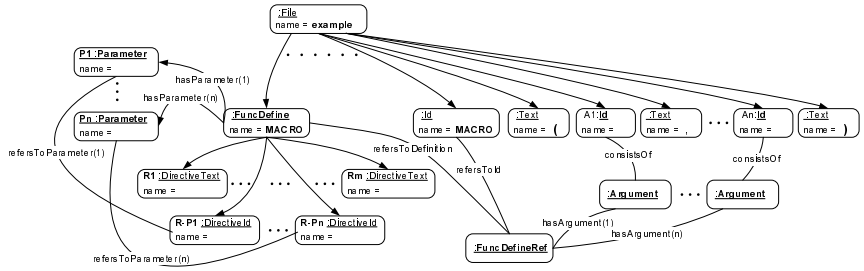

Figure 2: Add parameter transformation - left hand side of the rule.

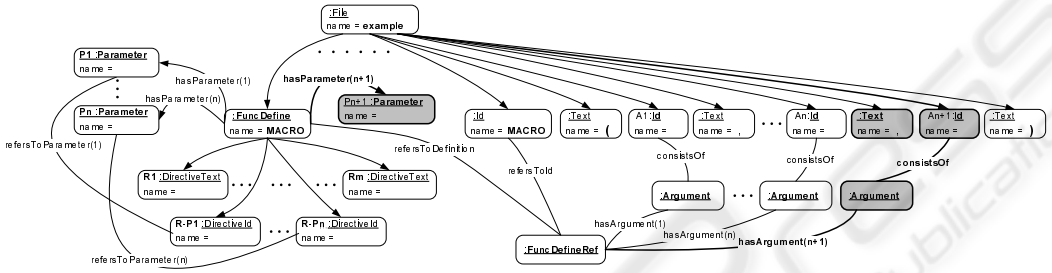

Figure 3: Add parameter transformation - right hand side of the rule (result).

The new parameter is included in the definition as the last parameter in the ordered association. Note that the new parameter is not automatically used in the macro body. Similar to object-oriented refactoring, the use of the new parameter has to be coded by hand. At the model level just one new node (*Parameter*) and one new edge are added (*hasParameter*).

Call Sites. In addition to the new parameter, each call site of the macro definition must be changed. According to the metamodel, each *DefineRef* object creates a link between macro definitions and macro calls. The *refersToId* relation indicates the position of the call, while the *refersToDefinition* relation indicates the called definition. With a type *A* transformation the macro definition usually has *DefineRef* objects, which are traversed. For each of them the pointed macro call is identified and at the end of the argument list a new placeholder argument is inserted. Finding an appropriate argument is easier than that for a typed language: in the preprocessor language everything is text, so any token is a valid argument. However it is recommended that an argument be inserted with a name which refers to the actual refactoring. The new argument is linked to the *FuncDefineRef* object via a new *Argument* object. For each call site three nodes and three edges are added to the model.

Implementation. The method has been implemented using an experimental tool setup including the Columbus reverse engineering framework and the USE model transformation tool. Experiments were performed on small but real life programs. Details are omitted from here due to space constraints. First results show the limitations of the approach in transforming large size programs.

3 RELATED WORK

In the previous sections we mentioned some research papers on refactoring, along with some of the available tools. The graph transformation approach to formal refactoring is also a well-known method. A remarkable summary of this topic can be found in the paper by Mens et al. (Mens and Tourwé, 2004). The graph representation of a program plays an essential role in the formalism. The two key issues here are preconditions and behaviour preservation. Bottoni et al. (Bottoni et al., 2004) use a similar formalism, the focus being on the coordination of a change in different model views of the code using distributed graph transformations. Most contributions try to be language independent, but for a refactoring to be applicable in case of real-life programs, language dependent details must be elaborated on. Although researchers have made good progress in this area, those in the industry sector use more or less the same solutions as before: language specific refactorings are implemented separately. Fanta and Rajlich (Fanta and Rajlich, 1998) report that transformations are surprisingly complex and hard to implement. Two reasons they give for this are the nature of object-oriented principles and the language specific issues. In our approach, instead

of using traditional graph transformation approaches we investigate language specific issues; we omitted *NACs* and used *OCL* to check conditions instead.

In the rest of this section we will concentrate on the preprocessor side. People working on C or C++ analyzers are confronted by the problem of preprocessor directives. The usual approach is to work on preprocessed code, or to recognize (partially handle) directives (Understand for C++ Homepage, 2008). Several preprocessor-related refactorings can be found in (Garrido and Johnson, 2002), which have no connection with the C language itself but with the preprocessing directives. The key aspects of such refactorings were presented at a conceptual level only, using source code examples. Dealing with directives is easier, when preprocessor constructs form complete syntactical units. Vittek (Vittek, 2003) introduced a tool which implements some preprocessor-safe refactorings on C++, while he acknowledged that there are unhandled cases caused by complex code constructions of the two languages. What makes this paper special are three points. The first is that directives are the main subject of refactoring at the model level. Secondly, the graph transformation approach is supported by *OCL*. Thirdly, we have tried to narrow the gap between academic research and industry by working on reverse-engineered program models.

4 CONCLUSIONS

Each modification of an existing program holds the possibility of making errors. So does refactoring, the technique for improving the quality of existing program code. In this paper a method was introduced to carry out program refactoring at the model level to assure the safety of modifications. The subjects of the refactorings were the preprocessor directives, which are usually omitted from C/C++ program analysis, however ignoring directives may lead to errors in analysis and transformation. As a demonstration of the approach, the add parameter refactoring for preprocessor macros was investigated at schematic and concrete level. Experiments were performed on reverse engineered models derived from several small, but real-life C/C++ programs. Future plans include the performance improvement by separating the design and validation and the execution of refactorings.

ACKNOWLEDGEMENTS

This work was supported, in part, by grants no. RET-07/2005, OTKA K-73688 and TECH_08-A2/2-2008-0089.

REFERENCES

Bottoni, P., Parisi-Presicce, F., and Taentzer, G. (2004). Specifying integrated refactoring with distributed graph transformations. *LNCS*, 3062:220–235.

Fanta, R. and Rajlich, V. (1998). Reengineering object-oriented code. In *Proceedings of ICSM 2008*, page 238, Washington, DC, USA. IEEE Computer Society.

Ferenc, R., Beszédes, Á., Tarkiainen, M., and Gyimóthy, T. (2002). Columbus - reverse engineering tool and schema for C++. In *ICSM 2002*, pages 172–181, Montreal, Canada. IEEE Computer Society.

Fowler, M. (2002). *Refactoring Improving the Design of Existing Code*. Addison-Wesley.

Garrido, A. and Johnson, R. (2002). Challenges of refactoring C programs. In *Proceedings of IWPSE 2002*, pages 6–14. ACM.

Gogolla, M. (2000). Graph Transformations on the UML Metamodel. In *GVMT'2000*, pages 359–371. Carleton Scientific, Waterloo, Ontario, Canada.

Gogolla, M., Büttner, F., and Richters, M. (2007). USE: A UML-based specification environment for validating UML and OCL. *Sci. Comp. Program.*, 69(1-3):27–34.

jFactor (2009). Homepage of jFactor. http://old.instantiations.com/jfactor/.

Mens, T. and Tourwé, T. (2004). A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139.

Mens, T., Van Eetvelde, N., Demeyer, S., and Janssens, D. (2005). Formalizing refactorings with graph transformations. *JSME: Research and Practice*, 17:247–276.

Opdyke, W. F. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, Urbana-Champaign, IL, USA.

Refactoring Catalog (2009). Refactoring catalog. http://www.refactoring.com/catalog/.

Roberts, D., Brant, J., and Johnson, R. (1997). A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263.

Slickedit Homepage (2009). Homepage of Slickedit. http://www.slickedit.com/.

Taentzer, G., Müller, D., and Mens, T. (2007). Specifying domain-specific refactorings for AndroMDA based on graph transformation. In *AGTIVE*, pages 104–119.

Understand for C++ Homepage (2008). Understand for C++ Homepage. http://www.scitools.com.

Vidács, L., Beszédes, A., and Ferenc, R. (2004). Columbus Schema for C/C++ Preprocessing. In *Proceedings of CSMR 2004*, pages 75–84. IEEE Computer Society.

Vidács, L., Gogolla, M., and Ferenc, R. (2006). From C++ Refactorings to Graph Transformations. In *Proc. ICGT'2006 Workshop SETRA'2006*, pages 127–141.

Vittek, M. (2003). Refactoring browser with preprocessor. In *Proceedings of CSMR 2003*, pages 101–110, Benevento, Italy.