Software Maintenance Methods for Preprocessed Languages

Ph.D. Dissertation

by
László Vidács

Supervisor
Dr. Tibor Gyimóthy

Submitted to the
Ph.D. School in Computer Science

Department of Software Engineering
Faculty of Science and Informatics
University of Szeged

Szeged 2009

Preface

So then it is not of him who wills, nor of him who runs, but of God who has mercy.

Romans 9,16

While this Bible verse above primarily tells us about attaining salvation, its truth may be discovered in everyday life. Our salvation is not simply the product of human will and effort, but a gift of God. In a similar way, God's undeserved goodness can be observed in many areas of a person's life and activities. In research work this may be even more apparent, where the outcome of a challenging initiation usually can not be predicted. Sometimes the result of a thoroughly performed, promising work becomes average, and at other times a sudden idea is accepted and honored. I believe that writing this thesis was made possible by the grace of God.

I should like to acknowledge several people who have contributed to this thesis in some way. First of all, I should like to thank my supervisor Tibor Gyimóthy, for providing interesting research topics and aims, and for his guidance over the years. Then I would like to express my gratitude to Árpád Beszédes and Rudolf Ferenc, with whom I discussed aspects of my research, for their valuable ideas and the time spent on joint work. My thanks also goes to my colleagues and friends, including Fedor Szokody, Judit Jász, David Curley, Péter Siket, István Siket, Patrícia Frittman and Richárd Dévai. I would like to thank Martin Gogolla as well, for his kind hospitality and support during the time I spent in his research group in Bremen. I am also grateful to András Kolozsi, Ferenc Havasi, Miklós Árgyelán and Zoltán Tasnády-Szeőcs, who are friends of mine; and especially to my love Katalin Horváth for her patience and support. Finally, I would like to thank my family for their continuous support and encouragement.

Contents

List of Figures

List of Tables

Listings

to my family with love

1 Introduction

Software development today has a considerable history. A large amount of effort has been invested in supporting the software development process. Many tools and programming IDEs have been developed and a large number of theoretical research papers have been published. The best-known methods and methodologies support the development phase of the software life-cycle. The need for software solutions in various economic areas, and the spreading rapid development techniques have enabled developers to produce thousands of lines of program code in a relatively short time. Due to the increasing number of software systems in operation worldwide, software houses and even end-users faced new problems about how to keep the software operating. The standard definition of the term software maintenance is as follows: The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. Correcting defects, keeping the software up to date with the changed environment and changing user requirements are costly activities. Any change in a system in the production stage costs much more than the same kind of change in an earlier phase.

Due to its nature, software maintenance requires activities other than the usual ones in the development phase. Program understanding is a crucial part of any maintenance tasks. The developers who want to modify the existing system needs both a high level, overall view of the system and detailed implementation level information. All this

information may be the result of the reverse engineering process. Reverse engineering is the process of analyzing a subject system in order to identify the system's components and their interrelationships and create representations of the system in another form or at higher levels of abstraction [18]. Impact analysis also benefits from the information obtained by reverse engineering. Change impact analysis helps in determining the affected parts of the system by a particular change. Thus, it provides support in cost estimation and test planning.

The real importance of source code based reverse engineering approaches (including ours) is shown by the fact that during software maintenance, the most reliable documentation is the source code itself. Other specifications, plans, documentations are becoming incomplete or inaccurate over the years of operation. In addition, outdated documentation may mislead the maintainer and create additional costs.

Although development and maintenance are strongly related, rapid development and good maintenance are often opposing notions. This phenomenon can be observed especially in the case of preprocessed languages. The usefulness of the preprocessor has been proven by many years of use by developers. Features improving productivity, like the flexible control over program configurations, the structured hierarchy of source files using includes, and the practical utility of the text-based macros (even parameterized), all provide reasons for the extensive use of the tool. An empirical study based on the analysis of commonly used unix software shows that preprocessor directives make up a relatively high 8.4% of source code lines on average [25]. The view taken by most is just the opposite when one has to assist in software maintenance or program understanding tasks: the presence of preprocessor directives is always mentioned as an obstacle [99]. The fundamental problem about preprocessing from a program comprehension point of view is that the compiler gets the preprocessed code and not the original source code that the programmer sees. In many cases the two codes are markedly different. These differences make program understanding harder for programmers and analyzers, and they can cause problems with program understanding tools. Reverse engineering techniques are often used when the maintainer has an insufficient knowledge of the system. The need for tool support is even greater in the presence of preprocessing directives, where the maintainer only sees the unprocessed code.

Our research efforts have concentrated on reverse engineering preprocessor directives. C/C++ source code analyzer tools frequently suffer from a common problem: the preprocessor directives are not part of the C/C++ language, therefore they need

a separate parser to analyze them. The problem affects a wide range of areas from calculating simple metrics through carrying out refactoring transformations to maintenance tasks like retrieving dependencies between software components and recovering the architecture of legacy systems. Without handling the preprocessor constructs, only partial and imprecise results can be obtained. Alas, preprocessor issues are often completely neglected by C/C++ analyzer tools, or at least, handled rather poorly. The basic idea behind our work was to respond to the current lack of general solutions and precise tool support. In the core of our work there is a detailed metamodel for preprocessing (in a reverse engineering context it is often called a schema). The schema describes the source code from a preprocessing point of view. It is an object-oriented model of preprocessor related language elements and their relationships. Our aim was to collect as much information as possible about the source code. Schema instances are concrete instantiations of the schema, produced by the analysis of the source code. Our tool implementations are part of the Columbus Reverse Engineering Framework. Based on the information obtained, maintenance tasks are supported in various ways. Preprocessor directives are usually taken in the account when refactoring C/C++ programs, but in the area of refactoring the directives themselves are rarely studied. In our recent work, we adapted a graph transformation approach to refactor directives at the model level. At the model level controlled and validated transformations can be performed, where model checking includes the checking of the metamodel itself.

Results were achieved in the area of change impact analysis, especially program slicing, which is an appropriate method for impact analysis. We introduced novel notions and methods for dependency-based slicing preprocessor constructs, using the Macro Dependence Graph and macro slices. On top of schema instances, macro slices are computed by our tools even for large programs like the Mozilla. Another novel result is the combination of traditional C/C++ slices with macro slices, providing a more complete dependency set for a specific slicing task. Hence we shall present a possible way to implement a preprocessor-aware C/C++ slicer, our implementation being based on existing tools (a well-known C/C++ slicer and our macro slicer) extended with a slice combiner. Traditional C/C++ and macro slices were evaluated by a set of experiments carried out using our tools.

After demonstrating how our work is related to software maintenance and development, and outlining research fields in which our results were achieved, the remaining part of this chapter gives a short summary on the structure of the thesis, and a summary

of the results achieved, also mentioning related publications of the author.

1.1 Structure of the dissertation

The dissertation is organized as follows. Chapter 2 introduces the background of our research. First, preprocessing aspects of reverse engineering C/C++ programs are discussed in detail. Next, the reader unfamiliar with the preprocessor can get information about its basic features, the standards and some implementation-dependent extensions of two mainstream preprocessor tools. Although the whole work is presented in the C/C++ language context, a separate section is dedicated to examples on how preprocessor is used within the environment of other programming languages, demonstrating the general nature of our work. After the introductory chapters, the thesis is divided into two parts along research results lines.

Part I - Modelling and refactoring preprocessor directives

The theme of the first part is the field of reverse engineering, with an emphasis on modelling preprocessor directives, and model level refactoring of macros. In Chapter 3 we focus on a metamodel (schema) called Columbus Schema for C/C++ Preprocessing. The metamodel plays a key role in the fact extraction and representation process, and serves as a basis for all other investigations here. The metamodel controls the building process of the model instances based on concrete programs. The two types (dynamic and static) of model instances are explained through various examples. Details on the implementation and the utilization of results are shown as well.

Chapter 4 presents our most recent work in the field of model level refactoring. A graph transformation approach is used with model checking, the USE specification environment served as graph transformation engine. Refactorings are performed on reverse engineered program models exported to the USE environment. The specialities of macro-related transformations and the necessary steps are shows through the add parameter refactoring on macros. Finally the tool architecture is outlined with the results of the performed experiments.

Part II - Slicing methods for change impact analysis

The second part presents our research results on slicing as a method for impact analysis. Chapter 4 contains background information on slicing in procedural languages. A small

motivating example is discussed and a real-world source code example is given in the next section, showing the need for our solution for macro slicing and for combining the two kinds of slicing. Afterwards, the utilization of macro slices and combined slices are depicted.

Chapter 6 introduces macro slicing, a novel notion for the impact analysis of macros. The basic theory and formal definitions are given for the dependency-based slicing of macros. The method is implemented using the existing metamodel, and experiments on a real-life software are reported. Chapter 7 is concerned with the idea of combining C/C++ language slices with macro slices. The theoretical background is outlined for the combined dependence graph of a C/C++ program, which contains information on C/C++ language elements and preprocessor macros as well. Combined slices are defined and illustrated in both forward and backward directions. The tool architecture and the slicing algorithms used are also presented. Then an evaluation of using combined slices on large number of open source software is presented.

Related research papers and tools are elaborated on in Chapter 8. The main part of the thesis is rounded out with some pertinent conclusions in Chapter 9. The appendix contains a summary of the thesis in Hungarian and English, and some details on the reverse engineering process.

1.2 Summary of research results

Our research work in the field of reverse engineering includes basic research on fact representation and metamodelling; and the utilization of extracted information in software maintenance, namely in program understanding, refactoring and slicing. Novel results are achieved in key, theoretical areas (metamodelling, definition of macro slicing, algorithms), but as good results were achieved in practice as well. The expected outcome of our theoretic work was always empirically evaluated with the several tools which have been implemented. Research results were successfully applied in several Hungarian and international research projects, both in industry and in the academic sphere.

The thesis result statements have been grouped into five contributions, divided into two parts according to the research topics. The relation between contribution points and supporting publications can be seen in Table 1.1.

	Contribution - short title	Publications
I/1	Preprocessor metamodel	[VB03] [VBF04]
I/2	Model level refactoring	[VGF06] [Vid09]
II/1	Macro slicing	[VBF07]
II/2	Combining slicing	[VJBG08] [VBG09]
II/3	Evaluation of slicing methods	[VBG09]

Table 1.1: Thesis contributions and supporting publications

Part I - Modelling and refactoring preprocessor directives

Contributions of the first part are related to preprocessor models extracted from the source code.

1. **Metamodel for the C/C++ preprocessor language**

   Our first result is the preprocessor schema (metamodel), which plays a key role in reverse engineering. The schema covers all preprocessor-related elements in a C/C++ source file, and also contains information on preprocessor operations (macro calls). To our knowledge this was the first publicly available general-purpose preprocessor schema. The schema consists of entities with attributes, and their relations, hence it is presented using the UML class diagram notation. A schema instance (model) is a graph that belongs to a concrete C/C++ program and contains all preprocessor-related information in a concrete form. From the schema instance the original source code, the preprocessed source code and all immediate states of the preprocessing process can be obtained. In addition, the schema describes both dynamic (configuration dependent) and static (configuration independent) instances. Therefore the solution is applicable for fully analyzing preprocessor usage at a fine-grained level. A programming API is developed for handling the graph from graph building to information extraction. To facilitate tool inter-operability and program understanding, the graph obtained can be exported to GXL and PPML (our XML format) as well.

   We implemented a preprocessor for building schema instances. The tool is called CANPP, it belongs to the Columbus Framework and it can analyze industrial size software projects with millions of lines of code. It is capable of building dynamic instances only, but this still offers a wide range of possibilities. The preprocessor

is designed to imitate the behavior of the GNU gcc/cpp and the Microsoft cl preprocessors, but intended to be fault tolerant, e.g. a missing include file will not prevent it from analyzing other parts of the program.

Our results were utilized in several industrial and academic research projects. All four of the following thesis contributions rely heavily on these results.

The schema and the related API is the work of the author. The implementation of model building is the work of the author, but the source code analysis technique and the building strategy of schema instances are based on the technology of the Columbus C++ Analyzer, hence these are shared results. The results of this contribution point are published in research papers [VB03, VBF04].

2. **Model level refactoring of macros**

   Model level refactoring has the advantage that it formally checks specific conditions, which is necessary when a high level refactoring has many concrete forms. Our first contribution is the set of viewpoints/steps for elaborating concrete macro-related refactorings with regard to the preprocessor metamodel. Based on the given criteria, we have presented a detailed discussion and elaboration of the refactoring named add parameter to macros. We applied a graph transformation approach by possessing left-hand side and right-hand side graphs.

   A special aspect of our work is that transformations are carried out on reverse engineered, real-life program models. We designed a tool architecture, mainly based on existing tools, capable for planning (important in elaborating concrete transformations), performing and checking refactorings on macros. We have implemented an exporter which transforms schema instances to an understandable form for an UML specification environment, in which the transformations are handled. During experiments we have found that the proposed tool-set is appropriate for middle size programs, but was good for validating preprocessor models and the metamodel itself.

   The above-mentioned contributions are the work of the author and published in research paper [Vid09]. Some notions used in this contribution point are published in an earlier work in C++ context [VGF06]. The used C++ metamodel is not the work of the author, the basic notions of model level transformations using the metamodel are shared results, while the elaboration and implementation of C++ refactorings are results of the author.

Part II - Slicing methods for change impact analysis

In the second part we integrate macro-related analysis to slicing.

1. **Macro slicing**

   The area of slicing is fairly diverse, and there exist lots of slicing methods and strategies. Their common attribute, however, is not to consider preprocessor macros as program points, the basic unit of slicing.

   Borrowing ideas from traditional dependency-based slicing, we introduced the novel notion of the Macro Dependence Graph. The dependency relation of the graph is derived from the macro call relation of directives and macro calls contained in the C/C++ program code. To ensure appropriate properties for slicing, dependency edges are colored in the graph. Therefore dependence graphs of complete software projects (not just compilation units or individual programs) can be built and used for slicing purposes. We defined both forward and backward type of macro slices, computable on the dependence graph. Using macro slices, complex macro-related questions may be answered in a change impact analysis context.

   Elaboration of macro dependency related definitions and the construction of the Macro Dependence Graph are the work of the author. Discussion of the notions of C/C++ and macro slicing, and the definitions of forward and backward slices are shared results. Results of this contribution point are published in research paper [VBF07].

2. **Combining C/C++ language and preprocessor slicing**

   The use of macro slices is limited to macro constructs, but the real advantage of our approach could be exploited if macro slices could be linked to traditional slices. Our novel result is the combination of traditional C/C++ slices with macro slices, giving a more complete dependency set for slicing. The connection points are the places in the source code where (initial) macro calls occur. These points are part of the Macro Dependence Graph, and the resulting tokens from the expanded macro call are part of the dependence graph of C/C++ slicing. We have composed a combined dependence graph, on which forward and backward combined slices are defined. Global forward and backward slicing algorithms are presented, as well. These contributions are significant improvements of traditional C/C++ slicing, and honored with best paper award. In the forward case

the improvement is more apparent, because C/C++ slicing would miss even the slicing criterion itself.

The construction of the combined dependence graph and the presented slicing algorithms are work of the author. The definitions of combined forward and backward slices are shared results. The results of this contribution point are published in research papers [VJBG08, VBG09]. Our paper, incidentally, won the *Best Paper Award* of the 16th IEEE International Conference on Program Comprehension in 2008.

3. **Experimental evaluation of slicing methods**

   Besides the theoretical results related to macro slicing and combining, we have performed experiments to evaluate the outcome of our proposed methods. The macro slicer tool was implemented on the top of the schema instances, since schema instances contain all information necessary for slicing, they play the role of Macro Dependence Graphs. Experiments were performed on the source code of Mozilla Firefox, which would be a hard task for a C/C++ slicer. We also outlined a tool architecture needed to implement a preprocessor-aware C/C++ slicer. Our implementation was based on existing tools, a well-known C/C++ slicer (CodeSurfer) and our macro slicer, and was extended with a slice combiner. Traditional C/C++ and macro slices were evaluated based on slicing time and memory consumption, the average and extreme slice sizes and the ratio of sizes. We found that macro slices were significantly smaller than static C++ slices on the same source code (the difference is larger in the case of forward slices). Despite being smaller, macro slices can provide a real improvement since they give precise information owing to their dynamic nature. We are not aware of any other similar report published in this area.

   The results regarding to the evaluation of macro slicing are the work of the author (macro slicing using schema instances, implementation of the macro slicer, experimental evaluation of macro slices). The results regarding to the evaluation of combined slicing are work of the author (tool architecture for combining slicing, implementation of the combined slicer, evaluation of C/C++ and macro slices), except the shared work on implementing the CodeSurfer plugin. Main results of this contribution point are published in research paper [VBG09].

2

Software maintenance in the presence of directives

2.1 Preprocessor issues in software maintenance

Four types of maintenance activities may be distinguished based on the aim of the changes. Traditional (corrective) maintenance focuses on fixing bugs in the code. The activities of other types of maintenance are also called software evolution. Adaptive maintenance means adapting the software to new environments. Updating the software to new user requirements is called perfective maintenance, while the aim of preventive maintenance is to help make the software more maintainable.

For any kind of modification of a software, the first necessary step is to get factual information about the system. While this is more trivial during the development, when all requirements and specifications are gathered, after years of operation a constantly changing software is more like a black box. Over the years usually new modules are added and the system is integrated with others. In many cases, key persons of the developer team are not available in the late maintenance phase. Hence tool support is of great importance for program comprehension.

2.1.1 Fact extraction and representation

The aim of the reverse engineering process is to extract information about the software to support further program understanding and analysis tasks [77]. In a wider sense, the subject of reverse engineering may be not only the source code, but specification documents. However, we shall follow the classical approach of source code-based reverse engineering. The process looks for so-called facts (or artifacts) in the source code, which may be any kind of usable information. The two key dimensions of the process are fact extraction and fact representation. The output of the process is a well-defined set of detailed information about the software, which is in an appropriate form for further processing for various purposes. Fact representation plays a key role since it determines the usefulness of the whole solution. The existence of several reverse engineering tools and the need for tool interoperability invigorated the design and publication of fact representation descriptions, the so-called schemas. Schemas are designed from various viewpoints, for various aims and degrees of granularity. Several efforts have been made towards a standard schema for the C++ language [36]. Enhanced reverse engineering tools have provided their exact schemas, like the Datrix schema [52, 8], the Bauhaus graph [22], the DMM system [66], the CPPX project [23] and the Columbus Schema for C++ [33, 35].

Data exchange [31] between various formats is frequently expected for complex problems. The GXL (Graph Exchange Language) is designed to be a standard exchange format for graphs [53, 51]. In particular, GXL was developed to enable interoperability between software reengineering tools and components, such as code extractors (parsers), analyzers and visualizers.

2.1.2 Preprocessor issues

C/C++ source code files are, in fact, not written in pure C/C++ language in the highest sense of the word. Feeding them directly to the compiler would result in compiler error messages. The source code for a compilation unit, understandable by the compiler is produced by the preprocessor [105]. Preprocessing is based on textual replacement rules controlled by so-called directives, which are not related to the C/C++ language. In other words, preprocessor operations are not aware of either the syntactical or the semantic elements of the C/C++ language. Hence, the expectation that preprocessor constructs form a well-formed C/C++ syntactical unit is unfounded.

The presence of the two forms of the source code (original and preprocessed form) is always mentioned as an obstacle in program comprehension. C/C++ source code analyzer tools often suffer from this familiar problem. As already mentioned, preprocessor issues are often completely neglected by C/C++ analyzer tools, or at least, handled rather poorly. Many automated tools designed to carry out program analysis and maintenance tasks work on preprocessed code, which results in mistakes at program points where directives are used. The need for accurate analysis and automated tool support for the above-mentioned activities is recognized by both the academic sphere and allot of industrial sector. A lot of effort has already been put into incorporating the preprocessor-related information into the processes which analyze the C/C++ language constructs, but so far with only moderate success. The problematic issues in preprocessing are typically the conditional compilation (`#if`), and the definition and usage of macros (`#define`). While there are usable tools for refactoring Java programs available, such tools for C/C++ have many problems caused by preprocessor constructs [114, 82]. Even a transformation as simple as a "rename variable" requires the analysis of preprocessor configurations [40].

The initial motivation behind our work was to extend the Columbus framework and its schema for C++ with preprocessor- related information. Instead of mixing the existing schema and C++ analyzer code with preprocessor-related analysis, we designed a separate schema for the preprocessor. The advantages of the separated analysis are clear: preprocessor information is ready to be used at any level in the Columbus system, but independent applications on the top of the preprocessor schema may be built for various maintenance tasks. Furthermore, the schema instances of a program are available as GXL exports and may be used by other tools as well.

The notions of static and dynamic analysis are present for preprocessor analysis as well. Usually static code analysis an means analysis without running the actual program, and dynamic analysis relies on data collected at runtime, depending on the actual input of the program. In our case, runtime means one particular run of the preprocessor. The run of the preprocessor depends on the environment and the command line options. The input of preprocessing is the collection of predefined and command line defined macros, and header search paths. Hence dynamic preprocessor analysis depends on these data, which determine the actual source code configuration. Static preprocessor analysis takes not only the actual configuration, but all other conditional blocks as well. In other words, in the case of a preprocessor, dynamic analysis means

a configuration-dependent analysis, while static analysis means a configuration independent one. Because even dynamic preprocessor analysis ends in the compile time of the embedding (C/C++) program, the result of both types may be incorporated even in a static (and naturally in a dynamic) analysis of the embedding language.

Similar problems arise when investigating the concept of backward and forward direction of slicing, and notions must be adapted and applied to the preprocessor. In the dependency-based slicing of C/C++ programs, the direction of the dependency relation is the same as the call relation, say. In the case of macros, the direction of a macro call is the opposite of the dependency relation. While the called function is dependent on the caller, in the case of macros the situation is just the opposite: the macro (caller) is dependent on the (called) macro definition.

In our work we show how these issues can be overcome with the help of the preprocessor schema, related notions and tools.

2.2 Preprocessor features

There are several C/C++ compilers that are commonly used, which have their own preprocessor implementation. Although together with the C/C++ language, preprocessor directives and features have also been standardized, several mainstream compilers have different and in minor details incompatible operations. In this section the main features will be introduced together with some usual and unusual examples of how the directives are used in real-life programs.

2.2.1 Standard preprocessor directives

Preprocessing is the first, separate phase of compilation. It employs its own token definitions, and the result of a preprocessing phase is a complete compilation unit, all preprocessor manipulations being performed on the source code before the compiler gets the code.

The latest ISO C++ standard [56] from 1998 contains a separate chapter on preprocessing. A year later, a new version of the ISO C standard was published [57], with similar content, but extended with the notion of variadic macros, which was already supported by main compilers at that time. Here, a small collection of preprocessor features is presented in an informal way. For exact definitions and complete references, please read the related standards.

Header files

File inclusion is a well-known feature. The included files are usually called header files, but there is no real limitation of the content of these files, and they may contain any kind of tokens. The file name may be given in three forms as shown in Listing 2.1

```
#include "stdio.h"
#include <stdio.h>
#include MACRO_CALL
```

Listing 2.1: Include types

The first form denotes a file belonging to the actual project, while the second denotes a standard library header. The difference is in the search order: in the case of the <...> form the file is searched on the standard library paths first. The third form contains a macro, which is fully expanded, and the final form must conform to one of the first two cases. Include directives may be nested, the limit of the nesting level depends on the implementation (and on the runtime memory). Recursion in include directives may cause problems, so the headers are usually protected by conditional directives (headers are called once only headers, while the conditional code is usually called wrapper ifndef).

Macros

Textual macro replacements provide the real essence of the preprocessor. Macros are used for several purposes and are usually combined with other constructs (includes, conditionals). There are two kinds of macros: object-like and function-like ones, the latter accepting arguments. A macro definition contains a macro name, optionally parameter list, and a replacement list. The replacement list may contain references to parameters, special operators, further macro calls and normal tokens. The two forms of macro definitions are shown in Listing 2.2:

```
#define OBJ_MACRO replacement_list
#define FUNC_MACRO(parameter_list) replacement_list
```

Listing 2.2: Two forms of macro definitions

Macro names are usually written in capital letters for the shake of readability in the place of macro calls. The actual macro definition of a particular macro name may be removed using the undef directive:

```
#undef MACRO_NAME
```

Listing 2.3: The #undef directive

Macros may be re-defined with a new define directive. According to the standard, if a re-definition occurs without an undef directive, then the new definition must be the same as the old one, but the most frequent implementations enable the re-definition, and just a warning is issued.

The scope of a macro definition is linear. It starts at the point of a definition, continuously persists through more source files according to the include hierarchy, and finally ends at the next re-definition, undef or at the end of the compilation unit. The macro expansion is a recursive process which results in a fully expanded macro. The expansion has specific rules. For more details, please check the IEEE standards. First the arguments are substituted (if there are any), together with the stringize (#) and concatenating (##) operators. During the expansion the replacement list is re-scanned many times while there are more macros to expand. The expansion depends on the place of the call, and not on the place of the definitions. This fact is important, because any identifier in the replacement list of a macro definition may become a macro call at a later point (by defining a macro for that contained name). We call this phenomenon the potential macro problem, as illustrated in Figure 6.5 of Section 6.3.

Variadic macros are a special kind of function-like macros, which accept a varying number of arguments (see example in Listing 2.4, note that a line break is inserted to the long line). Currently this feature is supported by the C standard only, but some kind of support exists in mainstream implementations.

```
#define VAR_FUNC_MACRO(param_list, ...) \
                    repl_list_1 __VA_ARGS__ repl_list_2
```

Listing 2.4: Example variadic macro definition

The ... parameter accepts any number of arguments, which are inserted in the place of the __VA_ARGS__ token. The named form of variadic macros is a GCC extension, which will be outlined later on.

Conditional inclusion

The set of conditional directives are similar to procedural languages, but the conditional expressions are different since they have to be evaluated in preprocessing time. This means that no variables and other C/C++ language constructs can be used: only constant expressions, macros and the defined operator can be used instead. The associated conditional block goes to the compiler only if the expression is not zero. A special feature of constant expression evaluation is that after the macro expansion and the replacement of defined operators, all remaining identifier tokens (except for `true` and `false`) are assigned a value of 0. This rule requires caution when writing conditional directives as an identifier which is thought to be a macro may cause the whole expression to be 0. As for conditionals `#if`, `#elif` and `#else`, only one conditional block is enabled, the others being skipped to the associated `#endif` directive. Conditional directives may also be nested.

Other directives

Line directives are important for tracking source code lines. This information is used, for instance, when the compiler issues a warning or error message. Line directives are produced by the preprocessor at specific places (for example, the return point of an include), but the programmer may also use it in the following form:

```
#line number filename
```

Listing 2.5: Example `#line` directive

where the file name is optional. The line directive causes the preprocessor to behave as if the following line was a particular line of the given file. Macro calls are also enabled in line directives, then the expansion must take the above form.

Pragmas contain special commands for the preprocessor or even for the compiler, because unrecognized commands are ignored by the preprocessor. Pragmas are used for stacking macros, controlling warnings and so on. The error directive causes the program to be ill-formed, with the error report given as arguments. The null directive also exists, but has no effect on the output.

2.2.2 Extensions, usual and strange constructs

Two well-known preprocessor implementations are the GNU GCC preprocessor [104][44] and the Microsoft (MSVC) cl preprocessor [76]. Neither fully support the ISO C99 standard, but both have some extra features beyond the standard. In this section we shall list some extra features and give examples of usual and unusual directive uses.

GNU extensions

The so-called once only headers are protected by guard macros, but this construction requires the header file to be opened every time it is included. The `#pragma once` directive causes that the file if opened only once. However, some experiments show that there is no measurable difference, most likely due to file caching.

GCC uses wrapper headers for exploiting old header functionality in new headers with the same name. The `#include_next` directive includes a header with an already used header file name, but the search for the header file starts from the previous position in the search path.

A named variadic macro is an extension, which allows the variable macro parameter to be named other than `__VA_ARGS__`. In addition, the concatenating operator has a special meaning in this case, so that the case of zero argument is also handled.

MSVC extensions

The `#pragma once` directive is implemented in this tool as well. The `/FI filename` command line options causes the given file to be included before the first line of the compilation unit. (Note that the same functionality is given by the -include GCC option.)

The MSVC implementation also uses macros to control compilation, such as the -GR option which defines the `_CPPRTTI` macro to enable the Run-Time Type Information feature.

There are several macros which are defined in case of specific circumstances. E.g. the `WCHAR_T_DEFINED` macro, which is defined when the /Zc:wchar_t option is used, or when `typedef unsigned short wchar_t;` is executed in the program code.

The `#import` directive is used to incorporate information from a type library. The content of the type library is converted into C++ classes, mostly describing the COM interfaces. This means that the directive is used by the compiler, and not just the

standard preprocessor. The `#import` directive creates two header files that reconstruct the type library contents in C++ source code. The primary header file is similar to that produced by the Microsoft Interface Definition Language (MIDL) compiler, but with additional compiler-generated code and data. The `#using` directive imports metadata into a program that will use the Managed Extensions for C++. Besides the stringizing operator (#), this implementation processes the `#@` charizing operator, which creates a character from its argument.

Configurations, computed includes

The possibility of creating source code configurations is one attractive feature of the preprocessor. The handling of complex configurations require all three main features at the same time: includes, conditionals and macros. Configurations belong to specific platforms, architectures, products, etc. A configuration is determined by the state of defined macros at the beginning of the preprocessing phase. Macros can be defined in many ways, and the actual configuration may be changed even without modifying the source code. Standard and environment macros are implicitly defined by the actual operating system and preprocessor implementation. Command line macros also control the actual configuration without changing the code. In addition, include paths can be modified, or the standard headers can be omitted using command line options, and in many cases includes are also computed based on macros.

Unusual preprocessor constructs

The variety of useful features may lead - in extreme cases - to unusual constructs which cause problems in program understanding.

The result of a macro expansion does not necessary form a complete C/C++ syntactical unit. This is the case when the macro body contains an opening parenthesis, but the closing one must be provided by the caller. This construction may be seem attractive because of its flexibility as can be seen in Listing 2.6.

```
#define strange(file) fprintf (file, "%s␣%d",
...
strange(stderr) p, 35)
     // ⟹ fprintf (stderr, "%s %d", p, 35)
```

Listing 2.6: Unusual macro definition

However it is recommended that one avoids these constructs [104].

The following code fragment is an excerpt from the Webkit JavaScript engine (see Listing 2.7). A block becomes a conditional block with an `if` condition depending on a macro call. Note that `ENABLE(CTI)` must be evaluated to true in compile time to enable the condition checking code, otherwise the block is always compiled.

```
// initialize local variable slots
#if ENABLE(CTI)
    if (!newCodeBlock->ctiCode)
#endif
    {
    }
```

Listing 2.7: Preprocessor conditional example

It is possible to create comments with macros. The idea is to paste two slashes with the concatenating operator. After calling the macro, the remaining part of the source code line will become a comment. This strange construction is used even in the standard headers of Microsoft Visual Studio. However, it should be avoided, on the one hand because it is suspicious, and on the other because it is not standard compliant. The comment processing phase is before the macros processing, so according to the standard, comments produced by macro calls are not treated as real comments and would cause a compile error.

2.3 The preprocessor in various languages

The scope of this work is not restricted to C or C++ language analysis. Although the preprocessor is closely linked to the C/C++ language, due to its flexible features it is used for many purposes in various languages. The possibility of making configurations, writing platform-dependent code fragments are considered attractive for programmers.

In this section we shall provide some examples of the diversity of purposes and environments in which the preprocessor is used. Some examples are several years old, but still show the extensive usability of the preprocessor. Our results were achieved in the natural environment of the C and C++ languages, but most of them (except the combination of C/C++ and macro slices) are applicable in the context of other languages as well. However, in subsequent sections in the remaining part of our work the preprocessor will be considered in its primary context.

Java

Java is designed without the preprocessor, and most of the programmers agree with this decision. The logic behind this is that java does not need platform specific configurations because the java code "runs everywhere". However in practice this is not the case. The mobile environment breaks this conception, as does the strong desktop integration, which needs to perform native operations [59]. A good example is the well-known Netbeans IDE, which recently used a preprocessor to deal with this issue. An interesting discussion on this topic can be found in the above-cited Javalobby forum from 2005 with the provocative title Does Java Need a Preprocessor? It turns out from the comments that many java developers actually use a preprocessor.

The preprocessor was specially designed for C, so there is a need for a Java version with more language specific support. The JPP project also recognizes the need for platform specific code inclusion [61], citing Eclipse as an example, which has to pass C pointers in the JVM, which may be int or long depending on the system it runs. It also names the missing `#line` directive as a reason for the preprocessor. The presented solution integrated the preprocessor into the build process of java programs. A more recent trend is to develop a java specific preprocessor. The javapp program [58] supports the C preprocessor-like syntax, but it is also extended with Ant integration and other features.

SQL

The Sqlpp project is a conventional cpp-like preprocessor taught to understand SQL (PgSQL, in particular) syntax specific features. In addition to the standard `#define`, `#ifdef`, `#else`, `#endif` cohort, it also has `#perldef` for calling arbitrary perl code [103]. (Note that this is not the usual sqlpp (sqp preprocessor), which processes the embedded sql statements in C programs.)

Perl

Besides the above-mentioned Sqlpp, another example for preprocessor use in perl environment is the ExtUtils::PerlPP (Perl Preprocessor) project [26]. It is written in perl and supports macro replacement and conditional inclusion with a C-like syntax.

The PICA (Perl Installation and Configuration Agent) tool also uses the preprocessor for configuration purposes [83]. In addition to the include and conditional directives,

it supports the `#perl/#lrep` pair of directives for executing perl code.

On various perl developer forums it can also be seen that the preprocessor is used for building perl programs.

Haskell

Similar to C and C++, large Haskell programs are not written in pure Haskell. Conditional compilation is extensively used with the help of the traditional C/C++ preprocessor. The `DEBUG` and `ASSERT` macros are also frequently called. A preprocessor suitable for Haskell was introduced in [118], demonstrating the need for Haskell related extensions.

PHP

CCPP (C Compatible Preprocessor for PHP) is an ongoing PHP project [17] that supports cross-platform PHP applications. It has some unimplemented features, but the aim is to be C standard-compliant. However, as with another languages, it has some specific directives like `#includephp` and `#literal`.

Python

Python developers also have an preprocessor-like solution for including files using the `##include` markup [84], and there is an ongoing project on Google Code [85] which implements preprocessor features with a different syntax.

Part I

Modelling and refactoring preprocessor directives

3

Metamodel for the C/C++ preprocessor language

3.1 Introduction

The initial motivation of our work was the fact that the C++ analyzer Columbus framework required preprocessed input files. Many software engineering tools choose this way because of time or effort constraints. A usual excuse for ignoring the preprocessor is that directives take up a really small part of program code. A thorough empirical study shows however that 8.4% of the studied source code lines consists of preprocessor directives on average [25]. A notable obstacle with the above-mentioned approach is that there is no exact information on token positions in the original source code. Developers of high quality reverse engineering tools at some point choose to invest in including preprocessor information in their solutions. This was the motivation of our work in the case of the Columbus framework; and similar decisions were made in the FAMIX system [69], for example.

Several researchers have been working in this area. A valuable contribution was made by Badros and Notkin [6], their framework allowing the user to write Perl callback functions to follow the work of the preprocessor (even in conditionally excluded code). The Ghinsu tool introduces coordinate mappings to describe macro calls [68]. The

GUPRO program understanding environment implements a fold graph that contains information for visualizing directive usage [63]). Similar to the latter two, most of these tools work only on one preprocessor configuration.

Schemas play a key role in the reverse engineering process, as already mentioned in previous chapters. We designed a preprocessor schema to thoroughly deal with the preprocessor. To our knowledge it is the first publicly available general-purpose preprocessor schema. We hope that this work (like the Columbus Schema for C++) will be utilized as a reference schema for other works. The schema also describes conditionally excluded parts and may be used to aid overall program comprehension and understanding code in real-world cases as well. Possible applications include macro call-graph extraction, macro-expansion visualization, include hierarchy extraction and so on. We have implemented a preprocessor which, besides preprocessing, is able to generate instances of the schema. Largely thanks to this, the mapping of the language elements to the original source code locations (e.g. where macro expansions are used) in Columbus has been improved .

To facilitate tool interoperability, the generated schema instances are also written in GXL format [53] so they can be used in software analysis, comprehension and maintenance tasks by other tools as well. Yet another application of the schema is in code quality assurance. Code containing preprocessor constructs may be checked against constraints and rules (in general, code with relatively simple macro complexity is better).

In the next section the preprocessor black-box problem will be introduced via an example, and we will also present our preprocessor schema. In Section 3.3 we give some example schema instances and ways they might be employed. Section 3.4 contains further details on our implementation, while Section 3.5 describes some applications of our results. Then, in Section 3.6, we draw some conclusions and mention some ideas for possible future research.

3.2 The Columbus Schema for C/C++ Preprocessing

Preprocessing means applying a set of low-level textual conversions on the source; the C and C++ language specifications ([56], [105]) have it in a separate part, and it is quite unrelated to the C++ language syntax. These text-based transformations are

hard to follow, and for reverse engineers the preprocessor is similar to a black box. The connection between its input and output is well-defined, but in concrete, real-life cases it may be hard to see precisely what is going on.

The schema is an object-oriented model of preprocessor-related language elements and their relationships. Object instances of the schema represent models of concrete source files, the resulting compilation units and the transformations being made by the preprocessor. This will then establish a connection between the original code and the preprocessed code. During the preprocessing of a C/C++ source our preprocessor tool builds the schema instance of the compilation unit, which represents both the source code and the preprocessing transformations applied on it.

The preprocessed output of a given source code varies due to the interactions of conditional directives, predefined macros and command-line defined macros. A configuration is the code belonging to one particular run of the preprocessor with a particular set of input macros. It is a far from trivial question of deciding how to handle these configurations in an analyzer tool, lots of other solutions deal only with the actual configuration.

To permit a wider range of information extraction we shall define two kinds of schema instances, with two possible ways of usage. The first is the *static instance* which does not depend on a given configuration (it will contain both true and false parts of an `#if` directive, etc.). The second is the *dynamic instance*, which is associated with one particular configuration, where conditional blocks ignored by the preprocessor are also omitted from the instance.

3.2.1 Motivating example

The advantage of having schema instances, which reveal the inside of the preprocessor black box, can be demonstrated with the examination of the following code fragment of `math.h` taken from the Unix standard library in Listing 3.1. One could start the investigation of this code as follows. The definition of the `__MATH_PRECNAME` macro depends on the `__STDC__` macro. Then `bits/mathcalls.h` is included and `__MATH_PRECNAME` is immediately undefined after this but, surprisingly, if we open the file `bits/mathcalls.h` in the source we can not find the text `__MATH_PRECNAME`. There are some questions raised by using this code. Is the macro `__MATH_PRECNAME` used between the `#define` and `#undef` directives, or is this definition unnecessary here?

```
#if defined __USE_MISC || defined __USE_ISOC99
...
#ifdef __STDC__
# define __MATH_PRECNAME(name,r) name##f##r
#else
# define __MATH_PRECNAME(name,r) name/**/f/**/r
#endif
#include <bits/mathcalls.h>
#undef __MATH_PRECNAME
```

Listing 3.1: Example code fragment from `math.h`

Does the compiler really get this piece of code for compilation? If it does, which of the two definitions is active? After a text-based search in the standard inclusion directory we find that `__MATH_PRECNAME` is present only in two headers. One of them is `math.h`, where it is part of a definition of another macro: the code fragment shown below in Listing 3.2 is in `math.h`, but it comes before the previously mentioned piece of code.

```
#define __MATHDECL_1(type, function,suffix, args) \
    extern type __MATH_PRECNAME(function,suffix) args __THROW
```

Listing 3.2: The searched macro definition from `math.h`

At this point we have to check whether or not this newly defined macro is present in `bits/mathcalls.h`, and we find that it is. But the following question still remains. There are two `#if` directives which come before the definitions of `__MATH_PRECNAME`. Is it possible that the compiler never gets this code? To answer this, we have to examine other macros to determine whether they are defined here, and what their values are. In general we can say that, to understand the code, the job of a preprocessor must be simulated by the programmer. Using our schema makes the whole procedure easier and a schema instance allows us to directly answer this and similar questions.

The outline of the dynamic schema instance of the example is shown in Figure 3.1 (only the relevant attributes are shown). As can be seen, `math.h` contains the definition of `__MATHDECL_1` (see node 10 in the figure). This definition is used at least once in `mathcalls.h` (28), which can be checked by navigating through the *FuncDefineRef* object (40). Also, `__MATHDECL_1` contains an invocation of the `__MATH_PRECNAME` macro (15), this invocation is connected (41) with its definition in `math.h` (22). It

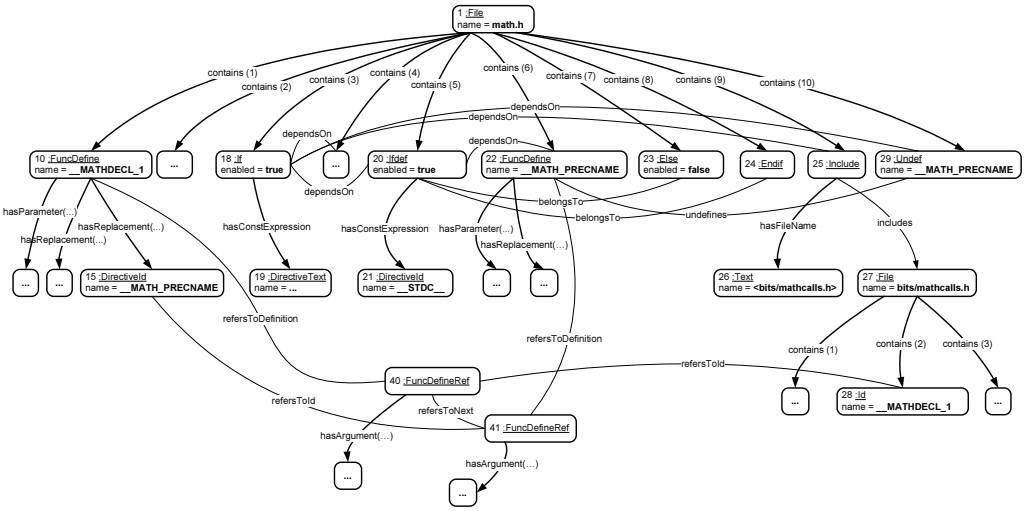

Figure 3.1: math.h: an example of a dynamic schema instance

can also be seen in the figure that the first `#if` condition (18) is enabled (evaluated to true), and also the `#ifdef` of `__STDC__` (20) was true, so the first definition of `__MATH_PRECNAME` was active (22).

Next, the questions listed at the beginning of this section can be answered in the following way. The `__MATH_PRECNAME` macro was used before it became undefined (so the definition is, of course, necessary), and the compiler gets this code fragment with the first definition. This is because the macro call (15) can be reached starting from *Include* at node 25, which comes before *Undef* at node 29.

As one might imagine from the example above, there are a number of typical questions about the preprocessing task. E.g.: Where is the active definition of a macro? Has this file really been included? Does the compiler get these lines after preprocessing? There are configuration independent problems as well, such as where all the different definitions of a macro can be found. With the help of our metamodel one can find a way of answering these questions and similar ones.

3.2.2 The preprocessor schema

The schema is presented using the UML Class Diagram notation [78]. Both static and dynamic instances are described by the same schema.

The UML Class Diagram of the preprocessor schema is given in Figure 3.2. The class *Base* is the abstract base class of all classes in the schema. Each element

that appears in the source file has a position, so (except for *File*, *DefineRef* and *FuncDefineRef*) all classes are descendants of *Positioned*.

The root of an instance is a *File* object. A *File* object can contain any number of ordered *Element* objects. *Element* is the abstract base class of elements contained in *File*. From the preprocessor's point of view a file consists of elements which can be either preprocessor directives or other text elements, so there are two specialized classes from *Element*: *Directive* and *Text*, the first also being an abstract one. Except for the text contained by directives, all textual elements in the file are represented by the class *Text*. The only parts of the source text of interest to the preprocessor are the identifiers (subclass *Id*), which are separate objects in an instance, because they may be macro calls. Otherwise the length and contents of text elements in one *Text* object is not determined by the schema, but by the strategy of instance building (it can be a preprocessing token or a longer sequence of characters).

Directives

Specialized classes of *Directive* correspond to the directive types. Most directives have various textual elements (like macro replacement) that are ordered lists of *DirectiveText* objects. The directives and their relations will now be described.

The *Include* directive inserts a whole source file in the position of the directive. An *Include* object includes a new *File* object, which is the root object of all elements of the included source file (this part is also completely expanded, and it may also contain further included files). The *hasFileName* relation between *Include* and *DirectiveText* associates the filename with the include directive. There are two different types of aggregation between *Include* and *File* in the static and dynamic cases (see the constraint in Figure 3.2). In the dynamic case when a file is included several times in a compilation unit, they require separate *File* objects with the whole subgraph because there may be macro definitions between the different include directives (or in the included file) which can influence the included file bodies even in one configuration. In this case the relation is a composition. In the static case the file which is included several times has the same content because the static instance is configuration-independent, so the one *File* object is shared among different *Include*s. To describe the command line forced includes (this means that the file given in command line is included before the first line of source file), the class *Include* has an attribute called *isExternal*. For an example on the include directive, see Figure 3.3 in Section 3.3.

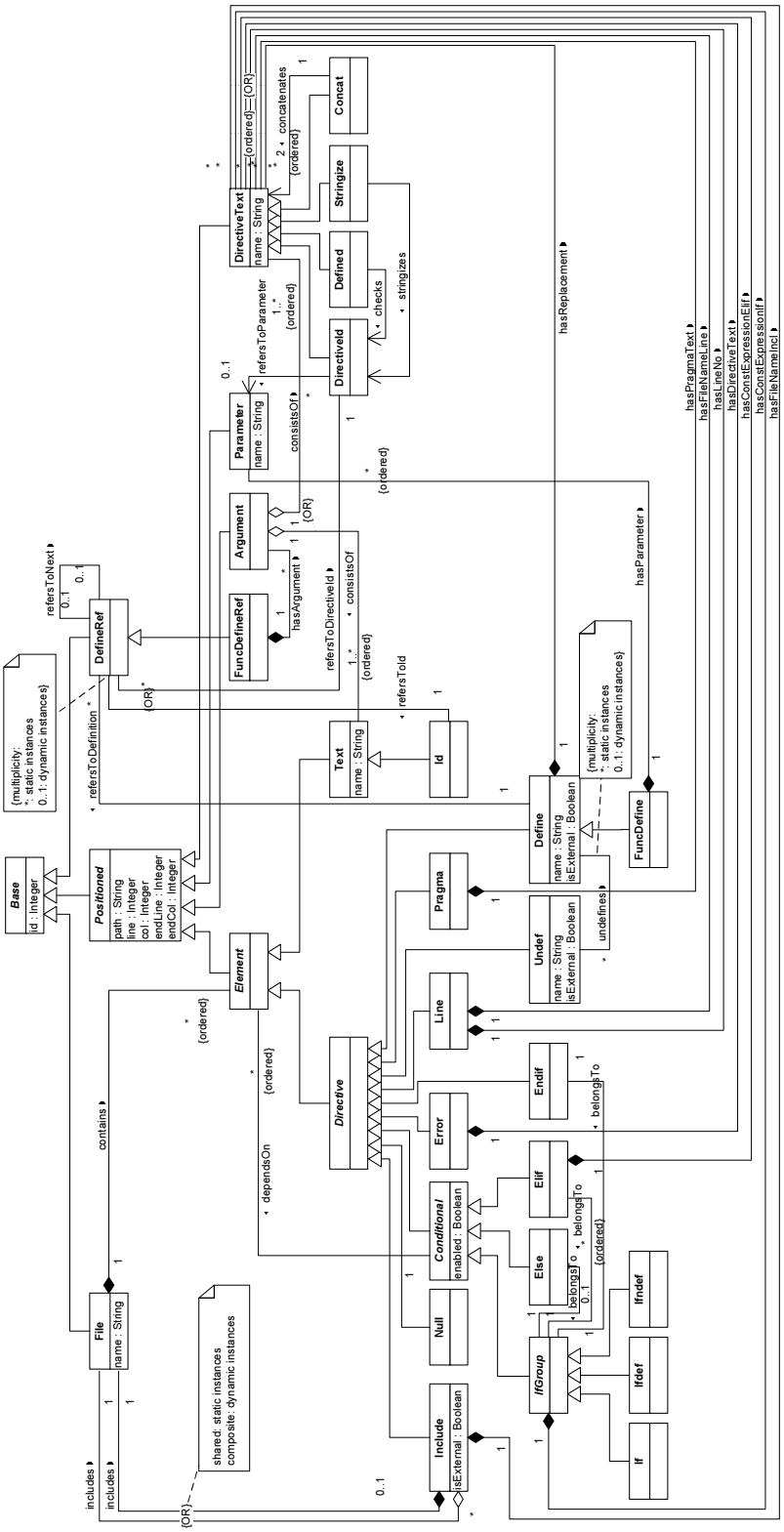

Figure 3.2: Class diagram of the schema

The *null* directive represents a hashmark followed by a newline, it does nothing and its class has no relations.

Conditional directives represent code blocks controlled by the conditional code inclusion (commonly known as conditional compilation [56]). *Conditional* is the abstract base class of conditional directives which determine conditional blocks. Conditionals *If*, *Ifdef*, *Ifndef* are derived from the *IfGroup* abstract class. The conditional inclusion is controlled by special expressions called integral constant expressions [56]. These expressions must be evaluated in the preprocessing phase of the compilation. The result of the evaluation is an integer which is treated as a boolean. They typically contain constants, macros and a special operator called *Defined*. The operator *Defined* has one operand and evaluates to 1 if the operand is defined as a macro name, and to 0 otherwise. Only *IfGroup* and *Elif* can have constant expressions. The conditional block is a list of sequential elements starting with an *If*, *Ifdef*, *Ifndef* or *Elif* and finishing before the matching *Elif* or *Else* or *Endif* pair of previous directives (note that conditional directives can be nested). Each *Conditional* object has a conditional block, which is linked to the directive using the relation *dependsOn*, because these elements depend on it. (There may be additional conditionals or included files in a block.) As regards an *If-Elif-Else-Endif* sequence, the code of a conditional block is included in the preprocessed output file only if this block is the first in the sequence, which has a conditional expression with a value of true. In this case the *enabled* attribute of the *Conditional* object is true, otherwise false (this attribute is relevant only in dynamic instances). To identify members of these conditional sequences the *belongsTo* relation is defined, so that *Elif*, *Else* and *Endif* objects can reference the appropriate *If* (or *Ifdef*, *Ifndef*) object.

Different configurations arise from different conditional blocks, but a normal run of a preprocessor produces only a single configuration (this is modelled with a dynamic schema instance). For a software maintainer it is important to see more (or all) configurations. Static schema instances treat all conditional blocks enabled, and therefore at the same time information can be gathered from more configurations. For examples, see figures 3.4 and 3.5 in sections 3.3.1 and 3.3.2, respectively.

An *Error* directive generates an error message and its use is usually combined with conditional directives. It has *DirectiveText* elements following the directive name, which are written out as an error message of the preprocessor.

The *Line* directive has two tasks: it generates line information for the compiler and

it redefines the `__LINE__` and the `__FILE__` standard C/C++ macros. *Line* has a line number and, optionally, a file name.

A *Pragma* directive is an implementation-defined control sequence for the preprocessor or the compiler (for example, to disable warnings or prevent multiple header inclusions). It has directive-texts which may contain macro invocations.

The *Define* directive is used to define preprocessor macros. Classes *Define* and *FuncDefine* will be described in the next subsection.

The *Undef* directive makes a previously defined macro undefined. It references only the corresponding *Define* object. One definition can be undefined zero or more times (only the first is accepted; the *Undef* directives which try to undefine not defined macronames are simply ignored). The relation present in the schema permits one *Undef* directive to reference multiple definitions, although in one configuration only zero or one definition can be referenced. Multiple relations are allowed only in the static case where all possible definitions (in different configurations) can be accessed (see the constraint in Figure 3.2). *Undef* also has an attribute called *isExternal* for the command line undefinitions of built-in predefined macros.

Macros

Macro definitions are represented by classes *Define* for simple, and *FuncDefine* for function-like macros. *Define* has a *name* attribute, and replacement text which is for replacing the macroname at the place of a macro call (this text is called a *replacement list*, and consists of *DirectiveText* objects). Definitions of function-like macros have zero or more ordered parameters. The formal parameter list is represented by objects of the class *Parameter* (the opening and closing parentheses and the commas separating the formal parameters are not present in the schema).

All textual elements inside directives are represented by the class *DirectiveText* (the text is stored in the *name* attribute). Identifiers have to be objects of the class *DirectiveId*. A macro replacement list may contain further macronames (*DirectiveId*) and it may also contain *Concat* operators (##). The *Concat* operator concatenates the preceding and the subsequent tokens and makes a new token.

The replacement list of function-like macros has some other special features. In the list a *DirectiveId* object may refer to one *Parameter* object: this *DirectiveId* will be replaced by the corresponding argument during a macro call. In this kind of replacement list, if the *Concat* operator concatenates a parameter then the parameter is

substituted before concatenation, and further macro expansions can be performed only after concatenation. The list may also contain *Stringize* operators (#). A *Stringize* operator must be followed by a *DirectiveId* (which is a parameter name), and during the replacement the operator creates a string literal from the actual argument. The two replacement list operators are specialized from *DirectiveText* because during preprocessing both produce new text from the arguments.

Macro invocations are represented by *DefineRef* objects. A *DefineRef* object refers to an *Id* (in simple text) or *DirectiveId* (in the text of directives) and links it to its definition by referring to the appropriate *Define* object (objects of *DefineRef*, being helper objects, do not represent any concrete source code element). One macro definition can be used (referred to) zero or more times. The invocation of a function-like macro (*FuncDefineRef*) has arguments which are objects of class *Argument*. Arguments are texts that are separated by commas. According to the place of the call an argument consists of one or more *Text* or *DirectiveText* objects, or their *Id*s because the argument may contain further calls. Parameter substitution occurs after every macro in the argument list has been expanded, but before other macros in the replacement list have been expanded.

The usage of *DefineRef* objects is different in the static and in the dynamic cases. In the static case a macro call (*Id* or *DirectiveId*) can refer to several definitions at the same time (with relation *refersToId*), and this way all possible definitions can be tracked, which can be important for a maintainer. In the dynamic case a macro name (*Id*) can be linked only with its active definition (its multiplicity is 0..1 in the dynamic case; see the constraint in Figure 3.2). At a given point in a source file the active definition of a macro is backward the first *Define* directive, which has no matching *Undef* directive (the included source files are also taken into account). The macro names in the replacement list of a macro may contain further macros (*DirectiveId*). This identifier may be associated with additional definitions even in the dynamic case. In the following example, shown in Listing 3.3, there are two expansions of macro `A` (lines 3 and 6). In the two cases different definitions of `B` will be active: in the first case it is the definition in line 1, and in the second case it is the definition in line 5. This difficulty with nested macro invocations necessitated the introduction of the *DefineRef* class and its *refersToNext* relation. When the replacement list or any argument contains further macro calls, the full expansion of a macro requires additional *DefineRef* objects, which are linked to each other with the *refersToNext* relation.

```
1 #define B 3
2 #define A k*B
3 A                  // ⟹ k*3
4 #undef B
5 #define B 5
6 A                  // ⟹ k*5
```

Listing 3.3: Scope of macro definitions

As we saw in the previous example, macro calls in a replacement list cannot be evaluated at the point of definition (macro call `B` in the replacement list of macro `A`). Once the macro expansion is started with an identifier, a list of *DefineRef* objects describes the first and the subsequent, generated macro calls. Each *DefineRef* object may refer to the next *DefineRef*, and each may be referred by zero or one object (the first has zero references). When a function-like macro is called, *DefineRef* objects for macro calls in arguments are included in the list before the further macro calls in the replacement list. The macro representation is explained in more detail below, and examples are also provided.

3.3 Usability of models

In this section some examples are presented on how some commonly used preprocessor features can be modelled using our schema. The details on static and dynamic instances are given below.

Dynamic instances represent exactly one configuration, while static instances let us see the overall code without any specific detail about a concrete configuration. We will illustrate this difference with an example of file inclusion. In static instances an included file name always generates the same *File* object, and each include directive refers to it. In dynamic instances every include has a different macro context and has its own *File* object, as outlined above in the description of the *Include* directive.

The dynamic and the static instances of the sample piece of code in Listing 3.4 can be seen in Figure 3.3.

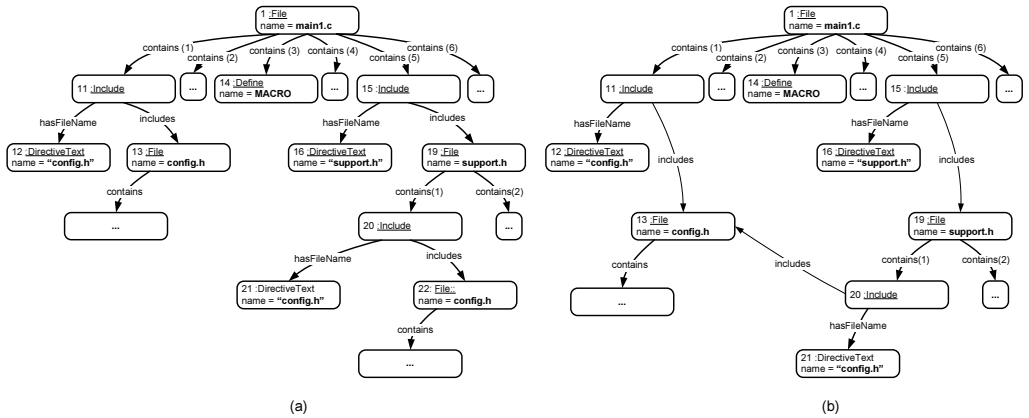

Figure 3.3: Dynamic (a) and static (b) schema instances of the include example

```
// file: main1.c
-------------------------
#include "config.h"
...
#define MACRO
...
#include "support.h"
...

//file: support.h
-------------------------
#include "config.h"
...
```

Listing 3.4: Example include directive

The same file (`config.h`) is included twice, but in the second case it is via another included file. In the dynamic case the *File* object is contained (via composition) in an *Include* object (11-13, 15-19, 20-22). In this example there are two *File* objects for `config.h` (13 and 22), this being caused by the different dynamic context of the two cases (e. g. macro definition 14 or the usual header protection construct *Ifndef–Define–*<*contents*>*–Endif*). On the other hand, static instances contain all conditional blocks regardless of the conditional expressions and contain all possible macro definitions and macro calls, so the two *Include* directives of the header `config.h` (11, 20) share the same object (13).

3.3.1 Static schema instances

To learn more about static instances, let us examine the following example (see Listing 3.5 and Figure 3.4 as well). The macro `LEVEL` is defined to be `1` by default (10), and there are two configurations: one for Unix and one for Windows. Both include supporting headers and redefine the macro `LEVEL` (21 and 28). After the directives the macro is called in a C if-statement. The macro call is linked via *DefineRef* nodes (40, 41, 42) with all the three possible definitions (10, 21, 28; using a pessimistic approach of the configurations).

```
// file: main2.c
-------------------------
#define LEVEL 1
#ifdef unix
#include "support_unix.h"
#elif defined WIN32
#include "support_win32"
#endif
if(LEVEL>2)...

// file: support_unix.h
-------------------------
#undef LEVEL
#define LEVEL 3
...

file: support_win32.h
-------------------------
#undef LEVEL
#define LEVEL 4
...
```

Listing 3.5: Include example for static instance

Definitions are also gathered from included files. In general, for a macro definition all possible macro calls can be seen, and vice-versa as well as all possible definitions of a macro invocation (in each configuration). Similar to macro calls, the *Undef* directives can refer to several definitions and one definition may be referred to by several *Undef* objects.

The details of static instances are not so well defined as those of dynamic ones. The strategy for building instances determines the final level of usability. In the previous

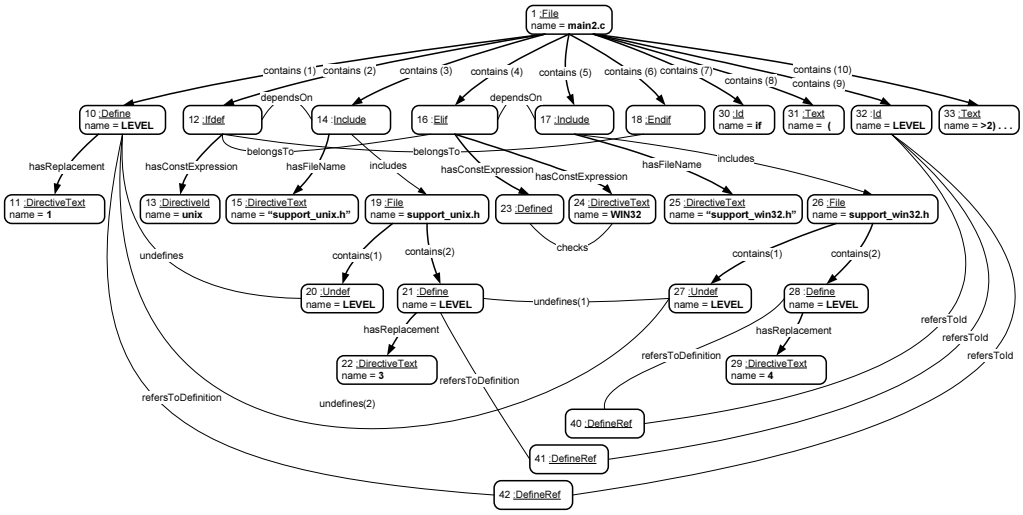

Figure 3.4: Static schema instance

example the *Undef* object 27 references two definitions (10 and 21). This is an example of a simple but safe building strategy. Actually, making a reference to definition (21) is not possible. It will never be present in any configuration because *Undef* (27) and *Define* (21) are always in different configurations. An optimized building strategy should filter these types of relations out. In the future it would be good to examine how we could implement more intelligent strategies for building static instances.

3.3.2 Dynamic schema instances

A dynamic instance is an accurate description of a configuration. It is more precise than the static one: the *Undef* directive, say, points to exactly one position, and the actual macro calls can be traced.

The most interesting part of a dynamic instance is the macro expansion, which makes use of a list of *DefineRef* objects. To see this, consider the following example code listed in Listing 3.6 and its dynamic instance shown in Figure 3.5. The macro `BASE` is defined as `200` (11). The function macro `ERR` (14) has two parameters (`type` and `place`; 15, 16), and its replacement text contains a function call using the two macro parameters. *DirectiveId*s (19, 23) in the figure below refer to the corresponding *Parameter* objects.

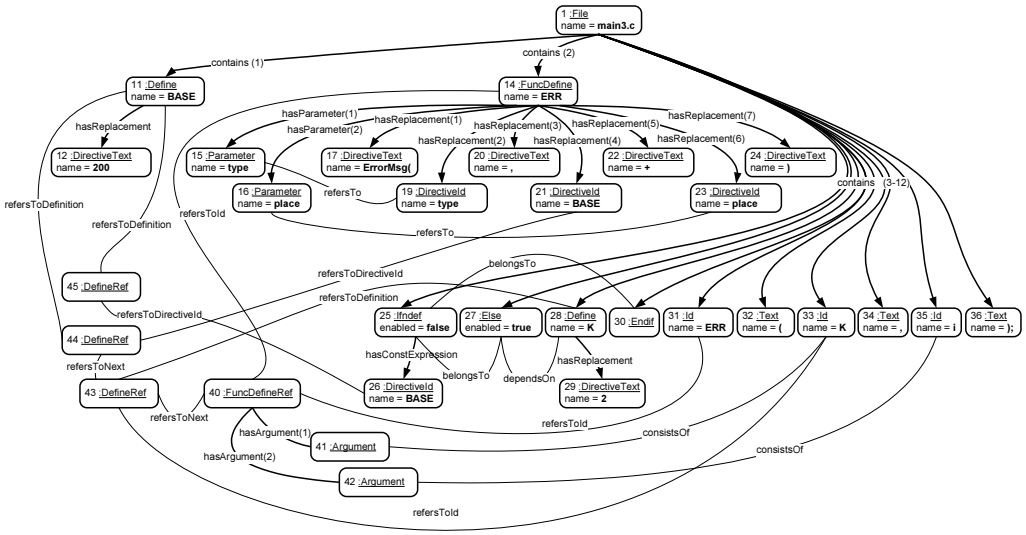

Figure 3.5: Dynamic schema instance

```
  main3.c:
-------------------------
#define BASE 200
#define ERR(type,place) ErrorMsg(type,BASE+place)
#ifndef BASE
#define K 1
#else
#define K 2
#endif

ERR(K,i);           // ⟹ ErrorMsg(2,200+i);
```

Listing 3.6: Dynamic macro expansion example

In the listed source code it can be seen that using conditional compilation, the macro `K` is defined as 1 or 2, depending on whether the macro `BASE` is defined. The dynamic instance contains only the *enabled* conditional block (27), so the definition of `K` to 1 is not present in the instance. Apart from the enabled block, the disabled directives are also stored in the instance together with their constant expressions, but without their blocks. Storing these constant expressions is useful. For instance, in the example the *Else* part is enabled, but to see why, we have to look at the constant expression (with a macro; 26, 45, 11) belonging to the *Ifndef* directive, which is not enabled. The relation *belongsTo* helps us find the matching conditional directives.

At the end of the example the macro `ERR` is called (31, 40) with `K` (33, 41) and `i` (35, 42) as arguments. The full macro expansion contains the objects 40, 43 and 44 in the list. *DefineRef* 43 links the first argument (macro call `K`) with its definition in the conditional block, so the actual argument will have the value 2 after substitution. The third *DefineRef* object (44) shows that, at the point of macro call `ERR`, the identifier `BASE` in the replacement list is a defined macro name. It is possible that later in the code the `BASE` macro may be redefined and the macro `ERR` is called. Then this call requires a new *DefineRef* object pointing to the new definition. Using the *DefineRef* and *Argument* objects, the final result of the macro invocation can be easily obtained from a dynamic instance.

3.3.3 How to get information out of the schema instances

In general, information extraction requires graph walks in the generated schema instances. In the following we will present some typical applications.

The extraction of the preprocessed file from a dynamic instance requires the following actions during the walk. Text elements not depending on directives are simply written out to the output. The directives are not written out, but instead all their effects are applied. This means that only the enabled conditional blocks are written out and the include directives are replaced with subgraphs. In addition, the macro substitutions are done by walking through each corresponding *DefineRef* object (at the same time argument substitution is performed and the required operators are applied).

Analyzing intermediate states of preprocessing helps us to better understand how the preprocessor works in a given situation. For instance, the levels of macro expansions and whether the included subgraphs are inserted instead of the include directives produce different intermediate states. To investigate nested macro calls and the levels of the substitution we can go through the list of *DefineRef* objects in a step-by-step fashion (relation *refersToNext*), exchange the *Id*s with the appropriate replacement texts and substitute the parameters if needed. This technique is very similar to the folding approach described in [63].

The include hierarchy of the compilation unit may be obtained from the static instance by simply traversing all the edges between the *File* and *Include* objects, starting from the *File* object of the input file.

As a last example we will show how the conditions under which a specific code line can pass through the conditional compilation and can be retrieved. Starting from the

point of interest in the static instance, one should walk through the *dependsOn* and *belongsTo* relations up to the root *File* object. The result is an appropriate combination of constant expressions of the traversed *Conditional* objects.

3.4 Building schema instances

Our model building preprocessor (CANPP) was implemented within the Columbus framework, along similar lines to that of the C++ Analyzer (CAN). CANPP is, on the one hand, an ordinary preprocessor and generates preprocessed program code, the usual `.i` files, which can be compiled by compilers. On the other hand, in the background a complete schema instance is built up in the memory and saved as a binary file. The tool works on exactly one preprocessor configuration, which means that only dynamic instances are produced.

An overview of our reverse engineering tools is given in Figure 3.6. The preprocessor is the interface through which the compilation unit in influenced by the environment. In the case of a compiler tool-chain, the environment contains several predefined macros. In order to produce preprocessed files, a complete standard library implementation must be available. The library search paths are also part of the environment of the compilation. We collected the necessary information to a `.ini` file, which contains standard library paths and predefined macros necessary for preprocessing. These sets of data are input for the CANPP tool, as shown in the figure. The preprocessor has command line options as well; among other things it is possible to define and undefine macros, or to add include paths to the process. The detailed list of command line options can be read in Appendix C.2.

When the necessary information is provided, the tool generates preprocessed (`.i`) files and preprocessor schema instance (`.psi`) files. To support whole program analysis, `.psi` files are merged to a linked `.psi` file. Hence whole software systems with several modules can be analyzed (for example Openoffice may be readily analyzed, with the help of the Columbus wrapping technique). The final step is to export the GXL or the PPML (Preprocessor Markup Language - our internal XML format, example code can be found in Appendix C.1) version of the graphs. In addition, using the API, schema instance files can be used for program understanding purposes; for example to implement the folding mechanism (shown in Section 3.5).

During the implementation our aim was to mimic two mainstream preprocessor

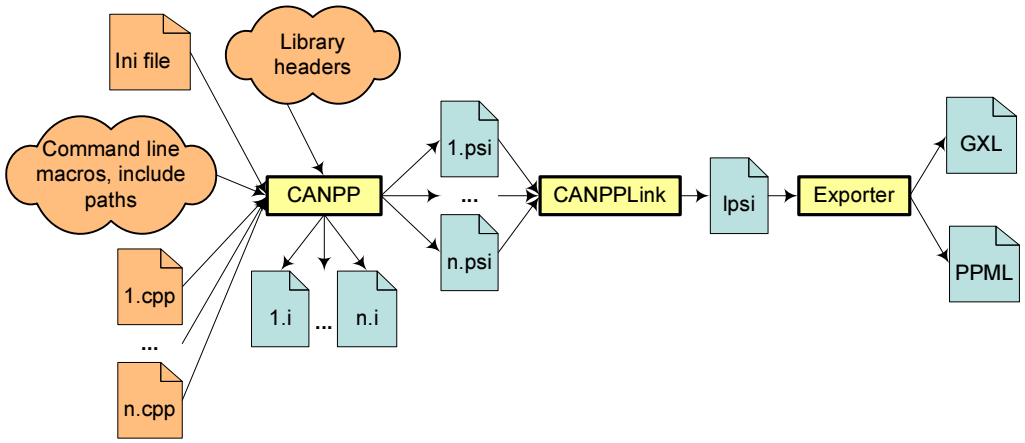

Figure 3.6: Preprocessor reverse engineering process

implementations, the Microsoft *cl* [76] on the Windows platform, and the GNU *cpp* [104] on Linux and Windows platforms. The test criterion was to obtain logically identical (except white-spaces, empty lines and `#line` directives) preprocessed output, which was successfully done on both platforms. Naturally, our tool is significantly slower than industrial preprocessor implementations that produce only `.i` files. A configuration utility has been developed both for the GNU and for the Visual Studio environments. The goal of the configuration process is to extract predefined macros and library paths (usually with versions) from the environment to create the `.ini` files. Currently the Visual Studio on Windows, and gcc on Windows and Linux platforms are supported environments.

3.5 Utilization of results

First of all, all contributions reported in this thesis make use of the metamodel and the extracted program models. The model level refactoring solution and the macro slices rely on these fundamental results. Many other solutions depend on the results of this contribution, which are not strictly part of the thesis but are in some way related to the work of the author. To demonstrate the importance of our results, we shall mention some important projects and tools.

3.5.1 Applications in the Columbus framework

The CANPP toolset is part of the Columbus framework. Preprocessor-related information is used in several domains.

SourceAudit The SourceAudit tool extracts various kinds of source code based quality attributes and checks coding rules [32]. Using this tool memory handling or code layout problems, and coding bad smells can be found automatically. The coding rules are extended with preprocessor-related ones; for example, macro names should be written in capitals. In addition, writing rules that check the original form of the code has become possible. For instance a limit checking of source code line length on preprocessed code is not exact because macro replacement usually increases line length, resulting in false positive warnings.

Macro expansion positions In some cases there is no need for the full mapping of unprocessed-preprocessed code; it is enough to know whether a particular C/C++ language element is a result of a macro expansion. For this purpose an algorithm calculates the output positions of the graph globally, and prints out a table for each source file with the intervals in the preprocessed file that are the result of macro expansion. These tables are then processed and handled, for example, when we want to highlight particular code fragments. This is one of the most frequently used applications of the schema.

Include dependency The textual output contains a header file dependency list of the current base input file. The list requires further processing, one application being to help in incremental parsing in the Columbus tool.

Macro folding Folding is a mechanism for displaying macro replacement information for programmers [63]. Debugging may be really hard to do when the compiler error belongs to a line where a macro is called. In this case the programmer sees only the name of the macro, but the expanded program code may be several lines long, which is hidden from the programmer. The folding mechanism allows the programmer to switch between the two views of the code (macro name and replacement) in a step-by-step manner.

The folding mechanism was implemented as a programming task of a BSc thesis [90]. A Visual Studio plugin (similar to the SourceAudit tool) analyzes the currently open source file and inserts control characters at the places of macro calls. Fold/unfold buttons control the display of the actual macro at the cursor position. The original macro name (folded state) is signed with right and left triangles: ▷MACRO◁, while the replacement (unfolded state) is shown between down and up triangles: ▽Replacement△, as shown in the following example:

```
#define A 2
#define B A + 3
B ⟺ ▷B◁ ⟺ ▽▷A◁ + 3△ ⟺ ▽▽2△ + 3△
```

Listing 3.7: Macro folding example

A screenshot of the plugin in Visual Studio together with the macro structure visualization is shown in Figure 3.7.

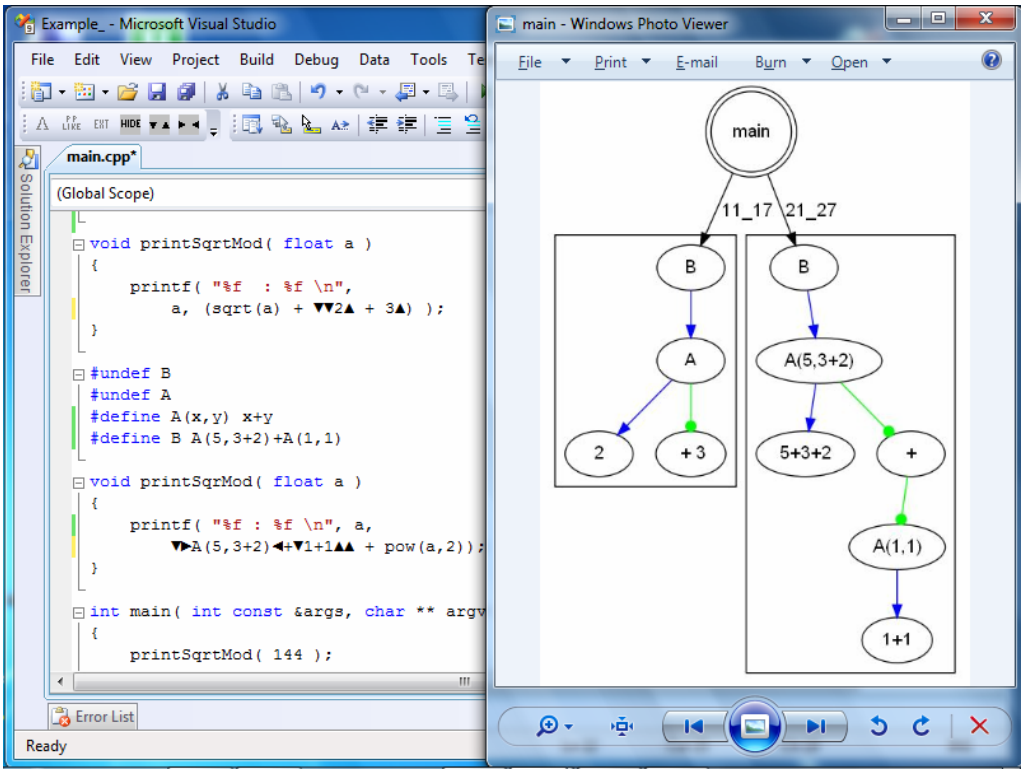

Figure 3.7: Folding in Visual Studio

3.5.2 Industrial and academic research projects

The results of preprocessor-related analysis have been used in several research projects. For instance a more than 5 million lLoc large software was successfully analyzed using our tool in a joint project with Nokia Research Center for compile time optimization. Another example here is a project co-funded by the EU. The aim of the OpenOffice++ R&D project was to analyze and improve the architecture of OpenOffice.org and the quality of its source code.

3.6 Summary

In this chapter we introduced the Columbus Schema for C/C++ Preprocessing. We showed through various examples that different kinds of program analysis, comprehension and maintenance problems can be overcome by using instances of our schema. The schema describes both dynamic and static instances for investigating the source code in a configuration dependent or independent way. Our implementation supports dynamic instances only, which still has a wide range of possible applications. The use of a standard notation and technology (UML, GXL) allows other reverse engineering tools to use the extracted information (for instance source browsers, visualizers and code-understanding tools), so it relieves researchers of the burden of having to write preprocessors for different purposes and allows them to concentrate on their own concrete research topic. The results of this work have been used in various ways from coding rule checking to visualization with the folding mechanism. In the future we plan to support static instance building and related applications in the area of configuration independent analysis.

The results of this chapter belong to contribution point I/1 (Metamodel for the C/C++ preprocessor language) and were published in papers [VB03, VBF04].

4

Refactoring at the model level

4.1 Introduction

In the past fifteen years refactoring has become an increasingly important technique for improving the design of existing code [79, 80, 110]. 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 model-driven development trend in software engineering, including the emphasis on iterative development. The strong need for tool support has resulted in an increasing number of refactoring tools, primarily for object-oriented languages like Smalltalk [91], Java [60], for C++ [95, 88], and for various kinds of languages, many of them listed in the refactoring catalog [89]. Refactoring is by definition 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.

In this chapter we will carry out the refactoring of C/C++ preprocessor constructs at the model level. We shall investigate the refactoring of reverse engineered program models derived from real-life C/C++ software, with an emphasis on the safety of the

transformations. Although refactoring in the C/C++ language is essential and affects many software developer companies, the tool support still needs 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 presence of the preprocessor introduces many obstacles [114]; even a transformation as simple as a "rename variable" requires the analysis of preprocessor configurations [40].

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 transformations. Naturally, preprocessor refactorings may be used later when composing C/C++ language refactorings as well. There are many points which need 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 safely perform a sequence of refactorings on a reverse-engineered program model. The importance of model level refactorings is emphasized by the fact that a high level refactoring cannot be performed, more special derivatives must be elaborated on instead. Here we propose a method for extracting program models from existing systems, planning concrete refactorings based on a high level description, performing them at the model level and verifying models and transformations. The transformations on the preprocessor metamodel will be described by graph transformations, which make them easy to understand and handle [45]. The Columbus and the USE systems were employed to implement our approach, which allows us to handle reverse-engineered program models and validate the transformations using OCL expressions.

This chapter is organized as follows. The next section contains our main contribution, including a short description of the important parts of the preprocessor metamodel, the graph transformation approach for refactoring, the reverse engineering process and the add parameter refactoring. Section 4.3 describes the implementation and the experiments. After, in Section 4.4 we draw some pertinent conclusions and make some suggestions for future study.

4.2 Refactoring on the C/C++ preprocessor metamodel

Refactoring is a way of improving the internal quality of the code [73]. Even when it is time to improve the maintainability or to correct the bad design of a former phase of 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 case major modifications may remain untested. Our intention here is to carry out controlled and validated refactoring steps (safe refactoring), which saves time and requires less effort.

In our study refactorings were performed on program models produced by the Columbus framework [34] using the preprocessor schema. The CANPP tool builds the graph representation of C/C++ programs, which includes information about the use of preprocessor directives, like macro definitions and calls, which conform with the preprocessor metamodel. The model-level transformations were realized via the *USE* system [47].

In this section graph transformation as a method for refactoring will be outlined, followed by a necessary description of the preprocessor metamodel. In the remaining part, the add parameter refactoring is investigated in detail.

4.2.1 Refactoring using graph transformation

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. In fact, there is a good correspondence between the notions and terms of refactoring and graph transformations [73, 71]. Mens et al. [74] give details on using graph transformations for refactoring and provide helpful illustrative examples. In addition to general approaches, efforts have been made towards applying this approach in domain specific environments as well [21, 67, 108]

We shall 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 will provide preconditions as *OCL* expressions. Despite the *NAC* concept being an integral part of graph transformation theory, we shall 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 directed, labelled, attributed graph where nodes, attributes and edges correspond to the metamodel that we shall employ:

- 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 coming 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 4.1).

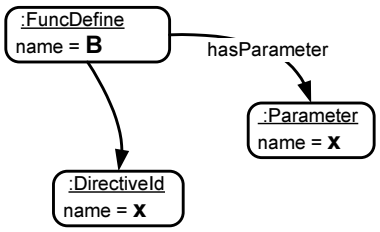

Figure 4.1: Object diagram-like notation of the graph

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 owing to the conditions of the transformations.

4.2.2 The preprocessor metamodel

We utilized the Columbus Schema for C/C++ Preprocessing as a preprocessor metamodel, which is represented in Figure 3.2 (in Chapter 3) as a UML Class Diagram. As the detailed structure of the metamodel is described in Chapter 3, only the necessary parts of it will be mentioned here.

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. Moreover, 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 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.

4.2.3 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 (in this sense it is not a classical refactoring as it is not behaviour preserving). The object-oriented version of this refactoring (add parameter to a method) is described in [37]. 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, then 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.

We recommend that the following steps should be taken into account:

- 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 may be automated at the model level, except for the first and the last one. In the following we propose viewpoints and aspects of planning concrete macro-related refactorings, and also highlight important parts of the preprocessor metamodel.

Alternatives In a rare case an alternative may be to get the required information from an existing parameter either by string concatenation using an appropriate string or other parameter, or by applying 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 lists), which in some cases make it unnecessary to add a new parameter. However a good reason for not making use of this type of refactoring is the long parameter list bad smell, which may be avoided by restructuring macros.

Preconditions The suitability 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__` and `__LINE__` macros
- Environment macros - defined by the compiler environment, e.g. `__GCC_VER__` for *GCC* and `MSC_VER` for *Microsoft Visual Studio*.
- Command line macros - defined as command line parameters, applied just to the actual compilation unit.
- 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. One is that 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, because argument substitution takes place before the further extraction of macros in the replacement text, hence the formal parameter name is replaced before any macro expansion.)

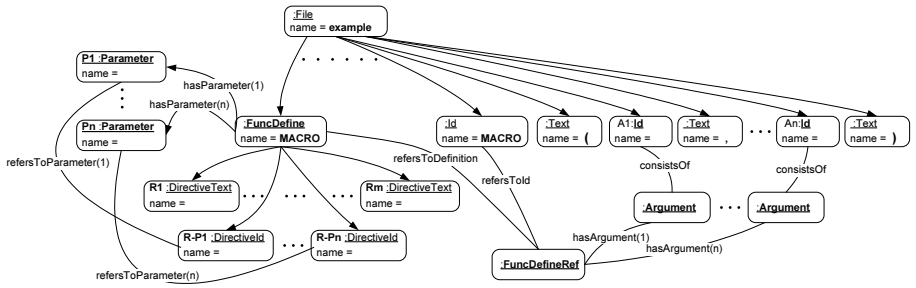

Figure 4.2: Add parameter transformation - left hand side of the rule

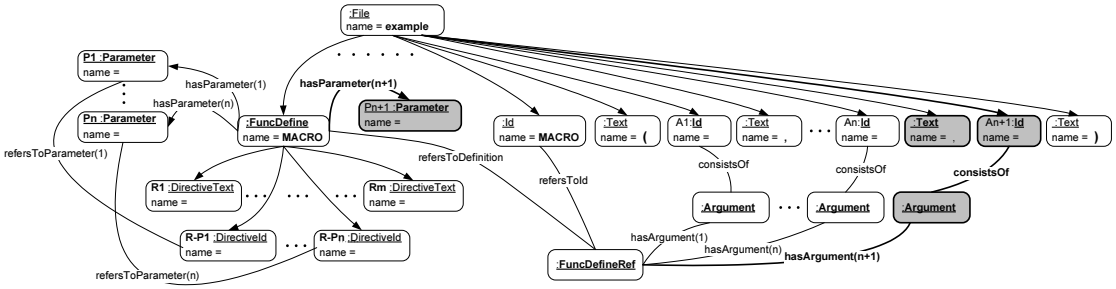

Figure 4.3: Add parameter transformation - right hand side of the rule (result)

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. These are:

A Function-like macros
This is the usual case: a parameterized macro is extended with a new parameter.

B Object-like macros
Because of the new parameter, the type of the macro definition has to be changed from an object-like to function-like macro. The two macro types are represented by different classes in the metamodel (*Define* and *FuncDefine*).

C Variadic macros
The variable parameter is always the last one, so the new parameter must be inserted before the variadic one.

The first is the basic case (type *A*), hence we provide a schematic graph transformation rule for the add parameter to a function-like macro in the two figures above. Figure 4.2 contains the left hand side of the transformation and Figure 4.3 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 4.3 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)
```

Listing 4.1: Schematic left hand side of the transformation

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)
```

Listing 4.2: Schematic right hand side of the transformation

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 refactorings, 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. In accordance with the metamodel, each *DefineRef* object establishes a link between macro definitions and macro calls. The *refersToId* relation indicates the position of the call, while the *refersToDefinition* relation indicates the called

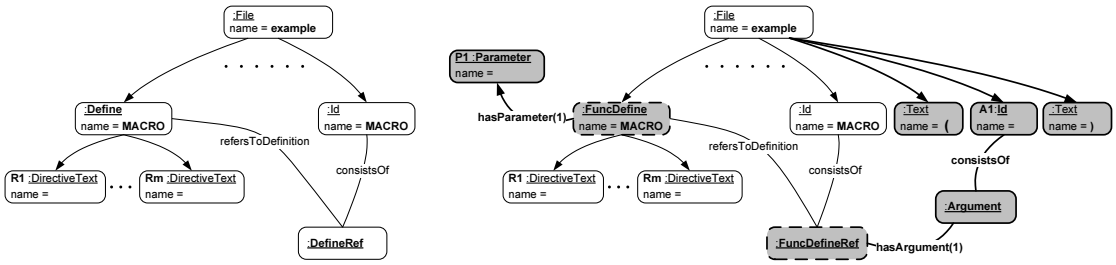

Figure 4.4: Add parameter to object like macro - left hand side and right hand side of the rule

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.

Object-like and variadic macros The second concrete type (B) of the refactoring is the extension of an object-like macro with a parameter. In this case there are more structural changes in the program. In Figure 4.4 both sides of the rule can be seen. On the left hand side there is a simple macro with its replacement list, while on the right hand side there is the function-like macro with a new parameter. The new nodes are shown in grey, added edges are in bold, while objects with a changed type are shown in grey and have been outlined with dashed lines. The new parameter changes the type of the macro, based on the metamodel from *Define* to *FuncDefine*, as well as the type of the reference from *DefineRef* to *FuncDefineRef*. A placeholder argument inside parentheses is added to each call site.

The last parameter of variadic macros (C) is called "..." (some implementations support the *name...* form). In this case, any number of arguments can be passed to the macro. The special token called $\_\_\mathrm{VA\_ARGS}\_\_$ will be replaced by the list of variadic arguments in the macro body. This type is similar to the first one as no type change is needed. However in this case the new parameter node and the placeholder argument are inserted before the variadic parameter and arguments, respectively.

4.3 Architecture

Refactorings are implemented using a reverse engineering framework and a model transformation tool. Besides carrying out experiments on small examples, we investigated the applicability of the method on several real-life open source programs. Experiments were performed using the object-like macro version of the add parameter refactoring.

Reverse engineering and transformations The reverse engineering of C/C++ directives is done using the preprocessor part of the Columbus system, while the planning phase of the concrete refactoring and the actual transformation are done using the USE system (UML-based Specification Environment [113]). An interesting feature of the USE system here is that it handles UML metamodels together with OCL constraints and queries in order to handle concrete models. Although USE was not originally intended for graph transformations, it can be used to realize graph transformations and it incorporates the advantages of OCL validation [16, 46]. We propose the following tool architecture depicted in Figure 4.5.

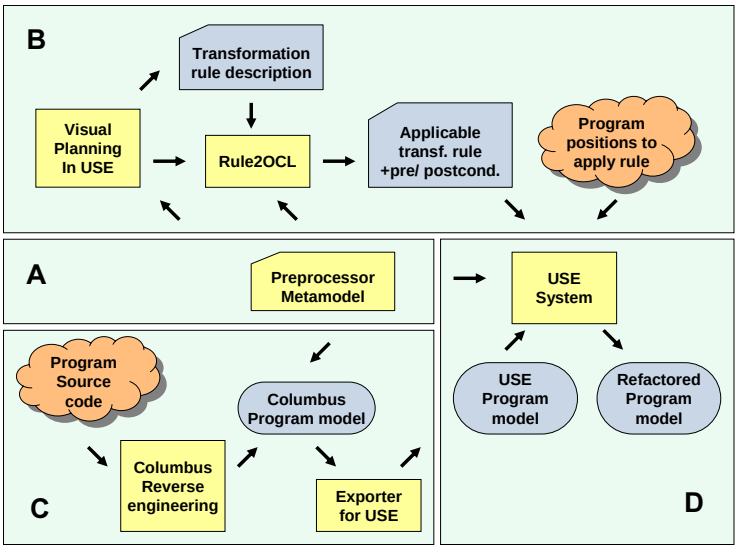

Figure 4.5: Refactoring architecture

The preprocessor metamodel is used in each part of the process (A). The transformation rule is designed by adjusting left and right hand side models in the USE system (B). Based on the models, a rule description file is created by hand (which is a straightforward step). The Rule2OCL tool uses the metamodel and the rule descrip-

tion to generate applicable rules. OCL pre and postconditions are also automatically generated. The Columbus tool produces the first program model (C). We implemented a new exporter that generates the program model in an appropriate form for the USE system. In this step elements of standard libraries are filtered out. The USE system checks to see whether the reverse-engineered model conforms with the metamodel (D). The rules are then applied at specified program points (in our case these points are automatically generated). The preconditions and postconditions on the refactored model are checked in each case and any inconsistencies are reported. A refactored object-like macro in the USE system can be seen in Figure 4.6. (Note that this is an implementation of the refactoring depicted in Figure 4.4.)

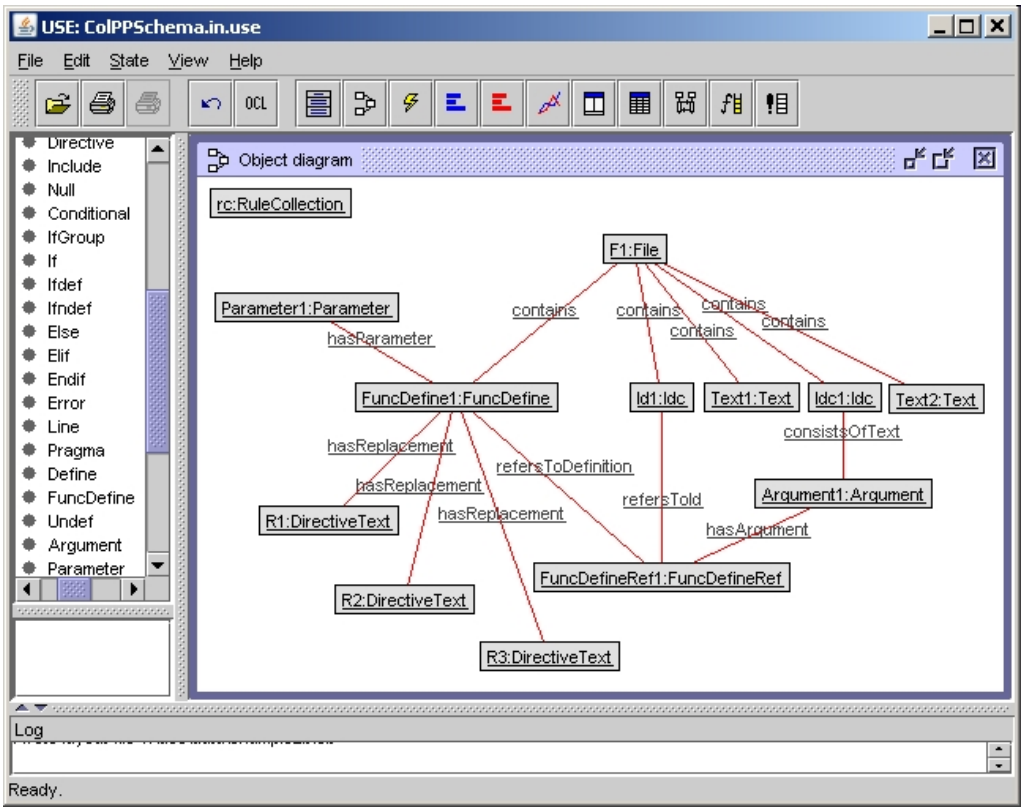

Figure 4.6: An object-like macro after refactoring in the USE environment

Experiments and limitations The object-like macro refactoring is implemented in two steps. The macro definition is changed and a parameter is first added, then each call site is extended with an argument. The macro definition on which the rule is

applied is the parameter of the rule. In our experiments we generated parameters to refactor all non-external object-like macro definitions.

The main properties of the refactored programs are listed in Table 4.1. Program sizes are given both in terms of lines of code and nodes in the program model. Both the number of object-like macros and the number of modified call sites are shown.

Program name	Size (lines)	Size (nodes)	Obj. macr.	Call sites	Time
time	1119	549	1	9	5s
wdiff	1364	644	4	18	6s
barcode	2807	999	23	48	25s
bc	9472	4296	47	264	48m
gzip	5997	4732	82	248	69m
diffutils	10124	6798	31	91	55m
Total	30883	18018	188	678	

Table 4.1: Refactored programs

The USE system automatically generates postconditions which are checked after the refactoring, the checking being the most time-consuming part of the transformation. It can be seen that the running time dramatically increases as the size of the model increases. We had to filter out standard libraries and we had to skip a small part of the postconditions.

Despite the above-mentioned limitations, the toolchain can be used for refactoring and checking small but real-life programs. However further work is needed to reach the applicability on huge industrial programs.

4.4 Conclusions and future work

With each modification of an existing program one may inadvertently create new errors. The same goes for refactoring, the popular technique for improving the quality of existing program code. In this chapter a method was introduced to carry out program refactoring at the model level to reduce the risk of creating new errors. The targets of the modifications were the preprocessor directives, which are usually omitted from a C/C++ program analysis. We presented viewpoints and steps for elaborating concrete macro-related refactorings based on the preprocessor metamodel. As a demonstration

of the approach, the add parameter refactoring for preprocessor macros was investigated at the general and concrete levels. Real-life programs were reverse engineered to obtain the models. Tool interoperability is assured by a new exporter which generates program models understandable for USE, based on the Columbus schema instances. The proposed refactoring architecture supports the visual planning of concrete refactorings based on high level descriptions; facilitates execution in a semi-automated way; and it supports constraint checking on models. The usability of the approach was demonstrated via successful experiments, where transformations were performed on reverse engineered program models derived from several small, but real-life C/C++ programs. In the future it would be helpful to investigate possibilities of completing the re-engineering phase by propagating transformations back to the program code itself.

The results of this chapter belong to contribution point I/2 (Model level refactoring of macros) and were published in paper [Vid09] and partly in paper [VGF06].

Part II

Slicing methods for change impact analysis

5
Background and motivation

Various program analysis methods and tools have been proposed to assist activities related to software maintenance. The handling of changes is a key issue in software maintenance, and an open question both in industry and the academic sphere [87]. Change impact analysis is the study of the ripple effect that is caused by a change in a large system. The goal of this process is to determine which parts of the system are affected by a particular change before the change actually takes place. This way impact analysis helps to determine whether the proposed change is safe or not.

A well-known method for aiding impact analysis is called program slicing. It is an analysis method for extracting parts of a program which represent a specific sub-computation of interest. There are a number of challenges which make the creation of practically usable program slicing tools difficult. Some of these are very general and have kept the research community busy for decades, while others are platform specific issues that are special for a particular programming language or family of languages or platforms. Our research deals with one particular issue, that of the *preprocessor*, in the context of computing program slices for C/C++ programs.

In the following chapters we will present the theoretic foundations and a possible way to implement a preprocessor-aware C/C++ slicer. The approach is based on the so-called *macro slicing* method. Essentially, a macro slice is a set of dependencies between macro definitions and their uses, which is fairly similar to other notions of

dependency-based slices. These macro slices are then combined with traditional language slices, thus providing a more complete dependency set for a specific slicing task. Macro slices are discussed in Chapter 6, while the details on combining macro slices with C/C++ slices are given in Chapter 7.

The rest of this chapter is organized as follows. First, an introduction to program slicing and its relation to our work are given in Section 5.1. In Section 5.2 we will justify the need for preprocessor-aware C/C++ slicers by providing motivating examples and application scenarios. Some utilization possibilities are outlined in Section 5.3.

5.1 Program slicing

Program slicing is seen by many as a very powerful technique that is applicable in various fields related to program comprehension and software maintenance. It is an analysis method for extracting parts of a program which represent a specific sub–computation of interest. Slicing was originally introduced by Weiser [119] to assist debugging, where a set of program points is sought for, which affect the variables of interest at a chosen program point, called the *slicing criterion*. The reduced program is called a *slice*. This definition is sometimes more precisely referred to as a *backward slice*, since – having procedural programs in mind – it associates a slicing criterion with a set of program locations whose earlier execution affected the value computed at the slicing criterion.

On the other hand, a *forward slice* is a set of program locations whose later execution depends on the values computed at the slicing criterion. Slicing can also be categorized as *static* or *dynamic*. In static slicing, the input of the program is unknown and the slice must therefore preserve meaning for all possible inputs. By contrast, in dynamic slicing, the input of the program is known, and so the slice need only preserve the meaning for the input under consideration.

Over the years, a number of algorithms to compute program slices have been developed; for an overview see [109, 121]. One of the most cited approaches is to apply a pre-computation step in which a representation of the program under investigation is first constructed, which captures the *dependencies* among program elements (for instance, data dependencies). This representation is called the Program (or System) Dependence Graph, whose basic form for static slicing and procedural languages was given by Horwitz *et al.* [55]. The nodes of this graph represent the program elements

(instructions), while the edges connecting them correspond to the program dependencies. The counterpart of this graph for dynamic slicing, the Dynamic Dependence Graph [4] includes a distinct vertex for each occurrence of a statement in the execution of the program on the input under consideration (called the execution history). Overall, the computation of a slice with these approaches means finding all reachable program elements in these graphs starting from the slicing criterion. In dynamic slicing, more recent results show that it may not be necessary to compute the whole program representation as the pre-computation step to make use of program dependencies [10]. Rather, slices may be computed *globally* by forward processing the execution history, in which case all possible slices are obtained. Alternatively, using a *demand-driven* approach only relevant dependencies are investigated in order to determine a particular program slice.

In many different program analysis fields researchers cite the preprocessor as an obstacle to implementing correct analyzes, *e. g.,* [40, 115]. Unfortunately, the situation is no better with program slicing. Alas, preprocessor issues are often completely neglected by slicing algorithms, or at least, handled rather poorly. Features like file inclusion or conditional compilation are sometimes handled in an acceptable way, but with macro expansion, for instance, it is a different story. The best that existing slicers can do is to mark those program points originating from macros and display this information on the screen.

CodeSurfer [49], for instance – which is probably the best-known static C/C++ slicer available today – displays information on macros appearing in slices, but is unable to include them in the slicing process itself.

A remarkable exception is the Ghinsu C slicing tool [68], which implements features for comprehending programs with preprocessor constructs, but unfortunately this project seems not being maintained anymore.

However, ignoring the existence of dependencies between preprocessor constructs and language elements may lead to serious errors in certain tasks where program slicing is applied. For example, in an incremental software development scenario, a change to a macro definition should be propagated throughout the system which will, in many cases, involve other macros and regular language elements as well. Impact analysis using slices that do not include preprocessor elements will be inaccurate and so potentially unsuccessful in situations like these.

5.2 Motivation

5.2.1 Motivating example

Developers usually have to make small changes during the system maintenance tasks, but in a large software system the effect even of a small change is hard to predict. Let us assume that the small program part to be altered is a macro definition. Our first motivating problem is to find the points of a C/C++ program which are affected by a modified macro definition. The modified definition may be used in (called from) other macro definitions, which may be called again from many points of the program (this is quite possible with macro slices). Next, the calls that use the definition are replaced and become part of the C/C++ language constructs. But these constructs may affect other parts of the program, which may be captured by traditional C/C++ language slices. In other words, the affected part of a program consists of *both* preprocessor-related elements and C/C++ program elements. The union of the forward macro slice starting from the given definition and the forward C/C++ language slice starting from replaced parts gives all the affected points. A small example source code which illustrates this is given in Listing 5.1.

```
1 #define ASSIGN(v) = v
2 #define SGN unsigned
3 #define DECLI(name, val) SGN int name ASSIGN(val);
4 DECLI(i,2)              // ⟹ unsigned int i = 2;
5 printf("%u\n",i);
```

Listing 5.1: Motivating example for combined slices

The slicing criterion for macro slicing is the macro definition in line 1. The corresponding macro slice contains lines 1, 3 and 4, while the macro call in line 4 is the link between the two kinds of slices. During preprocessing, the macro call `DECLI(i,2)` is expanded to `unsigned int i = 2;`, which is a C/C++ program element. The replaced macro is the slicing criterion for C/C++ language slicing, and the language slice contains lines 4 and 5. The combined slice contains all lines of the example code except line 2, which means that changing the macro definition on line 1 affects four lines. A failure to identify these additional dependencies may cause a problem in a change impact analysis, for instance.

The procedure of combining slices works in the other direction as well. Listing 5.2 lists the previously shown example code after the preprocessing phase.

```
1
2
3
4 unsigned int i = 2;
5 printf("%u\n",i);
```

Listing 5.2: Motivating example source code after preprocessing

The macro definitions are hidden from the compiler. Let the slicing criterion contain the variable `i` in line 5. The C/C++ backward slice algorithm does not know about macros as the slice contains lines 4 and 5 only. Using the fact that line 4 comes from macro replacement, a backward macro slice can be computed on line 4, which contains lines 4, 3, 2, 1. The combined backward slice contains every line of the original example, instead of two lines of the C/C++ slice. An example where this can cause a problem is when this additional data is not available in a debugger and the user is unable to track down to all the possible causes of an error which is being debugged.

5.2.2 Real world example

How useful combined slices can be is illustrated by the following example taken from the *flex* subject program of our experiments section. Let us assume that a new functionality is added to our software system and that we have to modify (among other things) the part of the program related to memory handling. It turns out that some part of the code to be modified contains a macro call in the original source. Using the macro backward slicing method, the macro definitions used can be accurately located in the code.

Let us assume that the macro definition `reallocate_integer_array()` found at `flexdef.h` line 686 is to be modified:

```
#define reallocate_integer_array(array,size) \
  (int *)reallocate_array((void *)array, size, sizeof(int))
```

Listing 5.3: Example macro definition from `flexdef.h`

Note that the "called" `reallocate_array()` is not a macro, but a C function. The following task is to build the whole program and then test it. Two problems may arise. The modified module compiles, but why do we have building problems for a totally "unrelated" module? And having modified the macro definition, which parts of the program now have to be tested?

In the case of modifying a C function, slicing can be used to determine dependent program parts, to give hints about affected files/modules, so one may select the appropriate test cases instead of performing full program test. In this case a combined forward slice on the altered definition should help. The macro slice shows that 31 toplevel macros are involved. The macro definition change is done based on one part of the program, but there are 30 other places where we have to do a test. An example path in the slice is when the definition is called from the `DO_REALLOCATION(dfa.c:261)` and `PUT_ON_STACK(dfa.c:269)` macros. The file `dfa.c` at line 308 contains a simple macro call, namely `PUT_ON_STACK(ns)`, but when the source is preprocessed, it is replaced by a do-while loop that is 358 characters long. One macro change goes through 31 points in the source, and for each a C slice must be computed, which finally shows that 8271 source lines may be affected. The 31 toplevel macros show where to check the correct macro usage, assisting us in build problems (sometimes in different modules). Then the full combined slice provides some hints on which part of the program is affected, which allows us to use selective retesting to reduce maintenance costs.

5.3 Utilization

The main idea behind the approach proposed here is the handling of macros. Generally speaking, the method is useful for the same purposes as C/C++ slicing: change impact analysis [14], program decomposition [39], software re-use [19, 122], debugging [3] and regression testing [13, 92]. Dependencies added by the macro slices provide a more accurate analysis and hence should produce better results. In the case of backward slices, the C/C++ slice is extended with macros, bringing source files into the slice that had not previously been taken into account. The special case of backward slicing was presented in the above example, where the backward slice was taken for a macro call to identify the macro definitions used. Forward combined slices start at a macro definition, which cannot be located using pure C/C++ slicing.

From a utilization point of view the preprocessor-related program constructs deserve

more attention. The backward direction can be used when the programmer encounters a macro call in the code, and neither the replaced value nor the used macro definitions are visible, which would help in the debugging process (the place of the compiler error is a macro call). This is true for program comprehension as well: the simple macro call is expanded to several C/C++ constructs like that shown in the previous example. As we have already seen, selecting the right test cases can be aided with our method as well.

The current implementation of the macro slicer works on just one configuration, which is analyzed by the C/C++ slicer. This helps keep the result synchronized, but also means that in general the toolset is not suitable for solving configuration-related issues. However, conditional directives usually contain macro checks (using the *defined* operator), which are included in a macro analysis. Thus the forward slice requested on a macro definition which determines the configuration (e.g. `#define USE_SMART_PTR`) will provide a hint about which part of the current configuration is configuration dependent. Unfortunately the macro call in a conditional directive is not matched to any C/C++ language element (see Section 7.3). A new dependency between conditionals and C/C++ elements would help. The current implementation of the macro analyzer contains a dependency relation like this, but it has not yet been used in macro slicing.

The data structure employed for macro slicing (introduced in [VBF04]) may be configuration dependent (as used in this work) or configuration independent. The latter has not yet been implemented, but in the future it may open the door for configuration independent macro slicing. The C/C++ part seems to be the harder problem though as the C/C++ slicer produces slices for just one configuration, but the configuration independent combined slicer should run the C/C++ slicer for every possible configuration and link/merge the results. Checking every possible configuration can be usually reduced to checking some key configurations, but at present a configuration independent slicer is just the subject of future research.

6 Impact analysis of macros

6.1 Introduction

In this chapter, we will focus on understanding macro usage. Usually, a macro related analysis is used to trace back the macro call to its definition. Although research tools implementing this feature (e.g. the folding mechanism of GUPRO [24]) already exist, the widely used debuggers still do not provide this information. A debugging tool support ends when the developer gets an error message from the compiler based on the preprocessed code. In many cases, it would be very useful to see the result of a macro call in the source editor. To answer questions like this, it is enough to analyze one compilation unit, but many software maintenance and program comprehension tasks also require inter-unit dependencies (covering the whole source tree).

During software maintenance tasks, developers usually have to carry out small but critical changes with no tool support for analyzing the impact of the change on the code, which may cause unforeseen problems. In the process of change impact analysis and change propagation, one tries to determine those parts of the source code which are affected by a change [86]. In particular, when analyzing the impact of changes in macros, we need to know all possible uses of a macro definition. In other words, it is needed to track the macro definition to all of its uses (macro calls) – as opposed to the other direction mentioned previously.

Our motivating question is hence the following: *Which parts of the source code are affected by a change in a macro body?* By affected points in the program we mean the places where the modified macro is called.

The intuitive method is to search the whole source tree using the *grep* tool so as to find all occurrences of the name of the modified macro definition. The resulting points need to be carefully examined because its name may occur in different contexts: as a simple macro call, in a macro body of another macro definition, or even in a comment. When the name is also contained in another definition, the search continues with the new name recursively. Thus, the whole process consists of many searches on many macro names, requiring careful consideration at each step. Apart from the large amount of effort and time needed to locate the affected program points without appropriate tool support, there are three main obstacles which makes the grep-like tools unsatisfactory here:

- *Includes and configurations.* Usually there are several configurations in the source code, and there may be a long distance between the macro definition and invocation because of many included files. It is usually not feasible to manually determine whether a search result is in the same configuration as the definition or not.

- *Macro redefinitions.* In spite of the fact that the standard forbids the redefinition of a macro name with a different macro body, it frequently occurs. The search result may belong to another macro definition, which will result in a false positive.

- *Hidden macro invocations using ## operators.* Using the concatenating operator with a macro parameter can result in a new macro invocation which cannot be revealed by a simple text search. The grep tool will produce a false negative error.

In this chapter, we will introduce a *novel technique* that addresses our motivating question. The next section contains the necessary terms and definitions for the analysis of macros. In Section 6.3, the macro slicing method is introduced. After a short discussion on macro and C/C++ slices in Section 6.4, we will outline our macro slicer implementation and report the results of experiments in Sections 6.5 and Section 6.6. In the last section we will give our conclusions and some closing remarks.

6.2 Definitions

In our approach, slices can be computed based on the structure of preprocessor macro calls and macro definitions. This is made possible by a special dependency relation defined on the call-definition structure. When a macro call is expanded, the initial (or toplevel) macro name is replaced with the replacement text of the macro definition. The definition may contain further macro calls which are expanded as well, so the full expansion of a toplevel macro may involve several definitions. Going the other way, the text of a macro definition may be used (through other definitions) in many macro calls. Separating the relevant replacements in a specific macro use is the task of macro slicing (presented in the next section).

When investigating preprocessor directives, the meaning of a static and a dynamic analysis is different from the usual meaning. The preprocessing phase takes place before the compilation and configurations of the program are controlled by an initial set of macros. A dynamic analysis uses runtime information based on one particular input. In the preprocessing case, the running time means the preprocessing phase which would be the compile time with the C and C++ languages. The input of the preprocessor is the set of macros which determines the actual configuration. We may say that the number of configurations is usually small or only a few of them are really important. Therefore, we perform a dynamic analysis of directives on one (or more) important configuration(s). This way we may miss some dependencies in other configurations, but this approach has two advantages: it is accurate because it is dynamic and it represents the whole software (or, at least, a key configuration).

The rest of the section contains the terms and formal definitions used in the analysis of macro calls. Many of the concepts described below are not restricted to the domain of dynamic analysis.

The following terms are used to formalize the macro replacements (see the example in Figure 6.1, the macro call results in `1 2`):

- *macro definition* – the place of the `#define` directive. The definition consists of three parts, namely *macro name*, optionally *parameters*, and *macro body* (also called the replacement list).

- *macro invocation* – the place in the program where a macro name is used (where the name is to be replaced with the macro body from the definition).

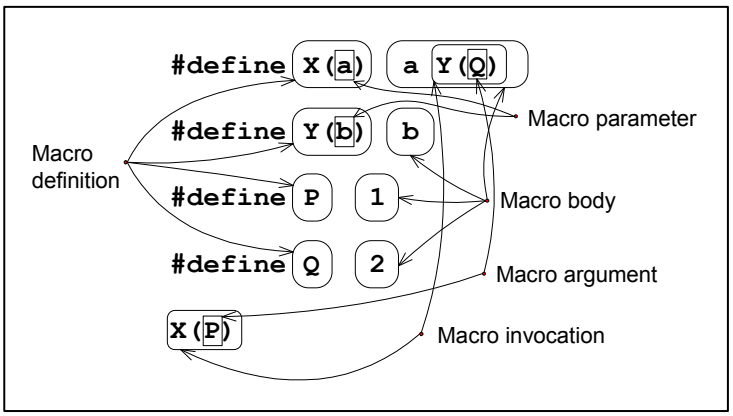

Figure 6.1: Example macro call

- *macro expansion* – the process of a single macro replacement, where macro arguments are also expanded and replaced.
- *full macro expansion* – all macro expansions which are necessary to get the final result of an initial macro expansion (including the macros in the re-expansion process of macro bodies).
- *toplevel macro invocation* - the starting point of a full macro expansion (a full macro expansion necessarily starts outside the `#define` directives).

Definition 1 *Let I be the set of all macro invocations in the given program.*

Definition 2 *Let D be the set of all* used *macro definitions in the given program.*

The fact of a macro call is represented by the $call$ relation between the two sets.

Definition 3 *Let $call : I \to D$, $call(x) = y$ if and only if the macro invocation x uses the macro definition y.*

The $call$ relation is surjective (D contains only called macro definitions) but not injective (one definition may be called from several places).

In the case of function like macros, a macro invocation may contain arguments . These arguments may also contain macro invocations, so we shall make the following definition.

Definition 4 *Let $arg : I \to I$, $arg(x) = y$ if and only if the macro invocation x calls a function-like macro and the macro invocation y is an argument of x.*

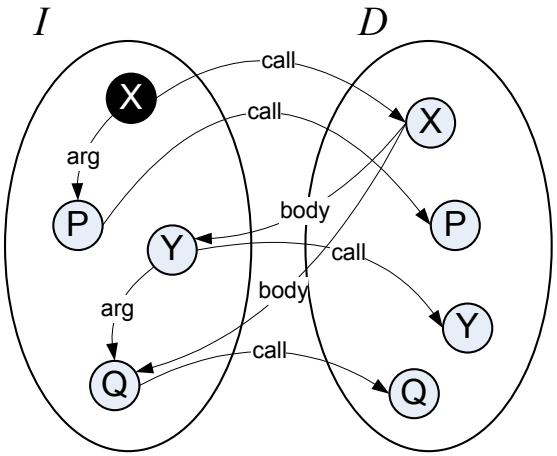

Figure 6.2: Macro sets and relations

A macro definition may contain further macro invocations in its body. This relationship is described in the following definition.

Definition 5 *Let* $body : D \rightarrow I$, $body(x) = y$ *if and only if the macro definition* x *contains macro invocation* y *in its macro body. (Note that when a macro body of* x *contains a function-like macro invocation with an argument which is also a macro invocation, then this latter invocation also constitutes a* $body$ *relation with* x*.)*

In order to increase readability and expressiveness, the sets can be contracted using the arg and $body$ relations (similar to a graph edge contraction). Let us construct a new set called MC that contains disjunct node sets that have elements from I and D. There are two types of new nodes. The first type is based on toplevel macro invocations (shown in black in Figure 6.2): each set contains a toplevel invocation and invocations which are in its arguments (a contraction using the arg relation). The second type is based on macro definitions: each set contains a macro definition and macro invocations within its macro body (a contraction using the $body$ relation). In Figure 6.3 there is a filled area for each element of MC. Formally, let

$$TI \subseteq I = \{x \in I | \neg\exists y \in I : arg(y) = x \wedge \\ \neg\exists z \in D : body(z) = x\}$$

be the set of toplevel macro invocations.

The elements of the new sets are defined using two sets according the two types:

$$MCI = \bigcup_{x \in TI} (\ x \cup \{y \in I | y \in arg(x)\})$$

$$MCD = \bigcup_{x \in D} (\ x \cup \{y \in I | y \in body(x)\})$$
$$MC = MCI \cup MCD$$

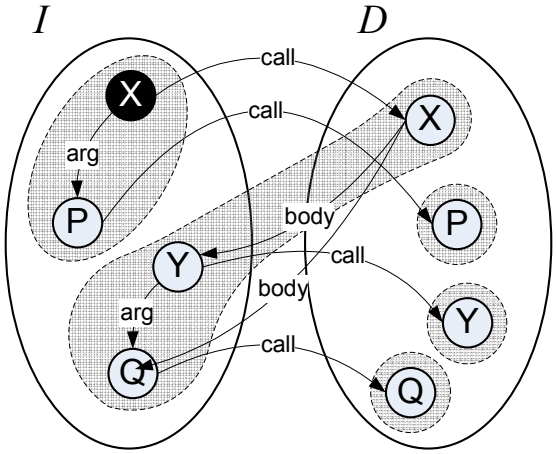

Figure 6.3: Elements of the *MC* set

The MC set is a subset of the powerset of the existing sets: $MC \subseteq \mathcal{P}(I \cup D)$ and all elements of I and D are included in one of the elements of MC. The call relation can be defined on MC as follows:

Definition 6 *Let* $call_m \subseteq MC \times MC$ *be a relation,*
$call_m(A) = \{B | \exists x \in A, y \in B : call(x) = y\}$, *where* $A, B \in MC$.

Macro dependencies can be defined based on the $call_m$ relation. Note that the dependency edge points in the opposite direction of that for the $call_m$ edge.

Definition 7 *Let* $dep_m \subseteq MC \times MC$ *be a relation,* $b \in dep_m(a)$ *if and only if* $a \in mcall(b)$, *where* $a, b \in MC$.

Figure 6.4 depicts the simplified set. Node sets (filled areas in Figure 6.3) are represented by their base nodes, as in Figure 6.4.

6.3 Macro slicing

Using the approach which restricts the slice criteria to used and defined variables we will define *forward* and *backward* macro slices. A slicing criterion is a pair $< p, x >$, where p is a program point and x is a macro definition or invocation.

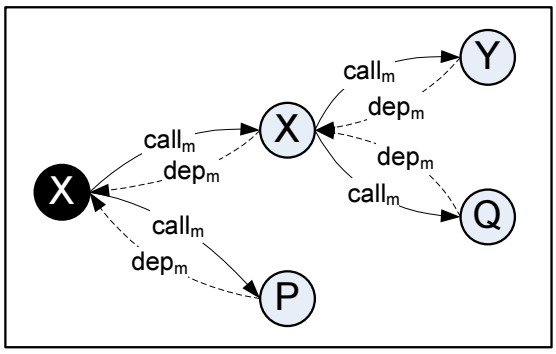

Figure 6.4: The $call_m$ and the dep_m relations on the simplified MC set

Definition 8 *The forward macro slice of a program based on the criterion $< p, x >$, where x is a macro definition, is the set of macro definitions and invocations that might be affected by the macro body of x.*

Definition 9 *Similarly, the backward macro slice of a program based on the criterion $< p, x >$ where x is an invocation consists of all macro definitions of the program that might affect the value of x at point p.*

Note that the forward slice of the criterion $< p, x >$ provides the answer to the motivating question stated in the introduction.

Slices can be produced based on the $call_m$ and dep_m relations using the definitions given in Section 6.2. The basic idea is to construct a graph where the nodes are elements of the MC set and the edges are constructed according to the $call_m$ and dep_m relations. Producing macro slices means solving a reachability problem starting from a given definition. Before constructing the appropriate graph on which slices can be calculated, the relations first have to be refined.

However, there is a problem caused by the fact that in a macro body every identifier is a potential macro name. The value of a macro depends on the place of the call, and not on the place of the definition. In the example in Figure 6.5, at the point of the definition of macro X identifier Y is a simple identifier, but it becomes a defined macro later on. At the point of the second invocation of macro X the identifier Y is a macro, so the full expansion of macro X starting from that point contains the expansion of macro Y.

Now we need to know which points are affected when the definition of Y is modified. A search based on the dep_m relation starting from the macro definition Y finds both

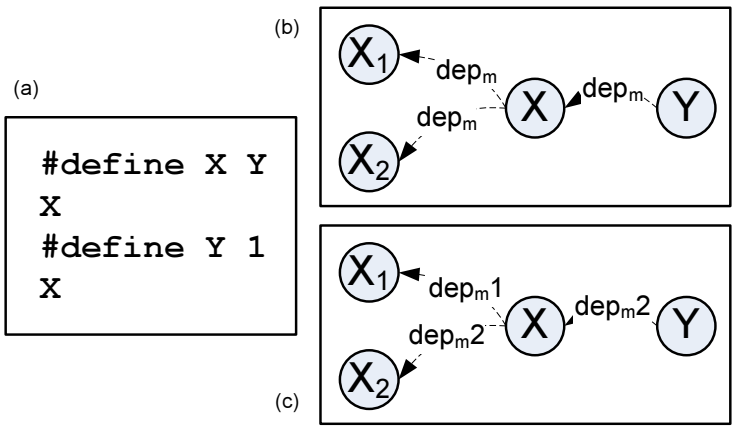

Figure 6.5: Potential macro problem: (a) program code (b) basic graph (c) MDG with edge coloring

X macro calls as dependent points, but only the point of the second invocation is really affected. In order to solve the problem of potential macro names which are later defined (macro re-definition causes the same situation) we have to distinguish the path on which a definition can be reached starting from the top level invocations. Full macro expansions have to be used to trace back macro replacements separately.

After the preliminaries let us now construct the Macro Dependence Graph (MDG). The nodes of the graph are the elements of the MC set and the directed edges are created from the dep_m relation. The edges are multiple edges because there may be more full macro expansions which have a common subset of dependency edges, but we have to distinguish them. Edge coloring is used to mark the edges that belong to a particular full macro expansion.

Definition 10 *Let $MDG = (MC, E, I, C)$ stand for the Macro Dependence Graph, where MC is the set of nodes (vertices) and E is the set of edges, $I \subseteq MC \times E$ is the incidence relation, for $\forall e \in E$ the $\{v \in MC : vIe\}$ set has two ordered elements, namely $a, b \in MC : vIa \wedge vIb \Leftrightarrow a \in dep_m(b)$, and $C \subseteq E \times \mathbb{N}$ is the coloring relation which assigns the same color to those edges which belong to the same full macro expansion. The E set contains multiple edges where each edge has a certain color, if several full expansions use the same edge. We use $dep_{mi} \in dep_m$ to denote the subrelation colored with i: $\forall i \in \mathbb{N},\ b \in dep_{mi}(a) \Leftrightarrow b \in dep_m(a) \wedge \exists e \in E : aIe \wedge bIe \wedge (e, i) \in C$.*

Slicing can be performed on the MDG. For a slicing criterion $< p, x >$ there is a node $k \in MC$ in the dependence graph which represents the macro definition x at the program point p. The forward macro slice contains exactly those program points which are reachable from k along colored edges in the graph.

Definition 11 *Let $< p, x >$ be a slicing criterion where x is a definition at program point p and $k \in MC$ is the node corresponding to x. Let Col be the set of colors which are used on dependency edges starting from k:*
$Col = \{c \in \mathbb{N} | \exists e \in E, c \in C(e) \wedge (k, e) \in I\ \}$.
The forward macro slice of the criterion is the set $S = \{y \in MC | y \in {dep_{m_i}}^t(k), i \in Col\}$, where ${dep_{m_i}}^t$ is the transitive closure of dep_{m_i}.

Because of edge coloring the search process of the slice elements needs to be modified: starting from the criterion, only those elements belong to the slice which are reachable via edges colored by those colors which start from the criterion node. A small example graph is given in Figure 6.5 part (c). The dependency edge colors are shown as numbers. The slice based on the definition of Y as a criterion contains the definition of X and the second macro invocation X_2.

It is important to note that the MDG is an acyclic graph when built from one compilation unit.[1] However, most software systems consist of several compilation units, and so the influence of a changed macro definition spreads to the whole system. Consequently, the macro call relations of individual compilation units need to be merged. Merging dependencies – in extreme cases – may bring cycles into the graph. To overcome this problem, each merged source file must have a disjunct color set. Such a merged graph is acyclic in the sense that there is no cycle with edges of the same color.

The backward macro slice can be computed on the same MDG if the edges corresponding to the $call_m$ relation are added with the appropriate coloring. Let I^{dep_m}, E^{dep_m} and I^{call_m}, E^{call_m} be the set of edges and incidence relations based on the dep_m and $call_m$ relations respectively. Let $MDG = (V, E, I, C)$ where $E = E^{dep_m} \cup E^{call_m}$ and $I = I^{dep_m} \cup I^{call_m}$. Forward slices are computed on dep_m edges while backward slices are computed on $call_m$ edges.

[1] According to the preprocessor standard, if a macro is under expansion and in the re-expansion the same macro is called again, this further call will not be expanded (the macro name remains in the replacement list instead).

6.4 Discussion on macro and procedural slices

In this chapter we applied the basic slicing principles to compute *macro slices* by constructing the *Macro Dependence Graph* (MDG). But some slicing concepts need to be reinterpreted within the scope of macro slicing, as we shall see in the following. In their first approach, Agrawal and Horgan introduced dynamic slicing by refining the static Program Dependence Graph using information taken from the execution history [4]. The need for the Dynamic Dependence Graph for constructing accurate dynamic slices was then demonstrated by the authors. Namely, a distinct node for each occurrence of an instruction was implied by the loops in the execution history. In the case of macro slicing the set of $call_m$ edges serves as execution history. The history of macro invocations can be reconstructed based on them (if a macro body contains more than one macro invocations, their order in history is the order of appearance in the macro body). Fortunately, there are no cycles in macro calls, so it is not necessary to create new macro definition nodes for each call.

Similar to other forms of slicing, we use the notions of *forward* and *backward* for macro slices as well. We should mention, however, that our choice for this terminology was rather arbitrary. In the case of procedural programs the slice direction is defined with respect to the *order of computations* in the program. But in the case of macros, the notion of "order" is less obvious as there are no "executable instructions" (consider, for example, the fact that the macro dependency edge points in the opposite direction to that of the macro call edge, while with procedural programs the control flow aligns with the control dependency). Furthermore, it is also meaningless to talk about data dependencies for macro slicing, since these may exist only between the actual arguments and the formal parameters, but the macro definition itself is not a part of the program, and hence the data dependency starts from the point of the initial call and necessarily ends at the same place.

6.5 Implementation

In this section we will examine the usability of macro slices defined above. To prove the practical usefulness of the concept we created a tool which identifies the affected points in the software based on the name of the macro definition. The tool works on the MDG of the whole source tree. Based on the name of the macro definition which is to be changed, the tool finds all points in the program which use that macro body

during the expansions. As an alternative solution, the software developer who faces the problem of changing a macro definition can use the power of simplicity: the *grep* tool. We will try to discover the effort involved finding the affected points in the source code by text-based *grep* searches made by hand. We will also investigate the limits of the grep approach and compare the process to the use of our search tool which works on the slice sets.

The basis of the macro slicer tool is the preprocessing schema introduced in Chapter 3. The CANPP tool is used together with the wrapper technique of the Columbus Reverse Engineering Framework [34]. The tool creates preprocessor schema instances from source files, and the schema instances are then linked at the system level. The result is the schema instance graph of the whole software, which contains information on macro calls as well. (Note that the schema instance contains information about macro parameters and arguments, so it is appropriate for discovering argument dependencies mentioned in Section 6.3)

Macro calls are modelled by *DefineRef* nodes in the schema. A *DefineRef* node connects a macro invocation with its definition (a *DefineRef* node is equivalent to a macro call edge). Full macro expansions are represented by so-called *call chains*. A call chain is a fixed order list of macro calls that belong to one particular full macro expansion. A call chain starts from the toplevel macro invocation; afterwards it contains the invocations in the arguments. In addition, the chain contains all invocations from the macro body recursively.

The MDG can be constructed based on these chains. Elements of I and D are contained by the schema instance graph, *DefineRef* nodes represent the $call_m$ edges, and one chain means one color in the MDG. The tool works on the schema instance graph without constructing the MDG. It requires the name of the macro definition which is to be changed (the slice criterion) as a command line argument, and looks up the definition in the graph. The definition contains references to chains that go through it. The tool walks through the chains back to the starting point (toplevel invocation), and in the meantime it writes out the affected nodes. Each visited definition during the walk is affected by the modification of the base definition, and eventually there is the toplevel macro invocation where the macro call started (demand-driven approach). Usually the developer is interested only in this last point of the chain, but the tool provides all of them. Note that backward traversal of the chain reaches nodes which are not part of the slice; these are filtered out or marked as members of the chain but

not included in the slice.

The slice data is written into a text file in the following form: the file consists of sequences, there being a sequence for each chain which goes through the modified definition. Each sequence consists of macro definition and macro invocation pairs until the start of the chain reached. The invocation and definition is printed out in an appropriate form to help in locating them in the source code. Details about the full path, source code line and column are also given so that commonly used programming IDEs like Microsoft Visual Studio or Eclipse can be used to quickly jump to the program points.

6.6 Experiments on large software

We performed some experiments to validate our tool on the Mozilla Firefox internet browser. The experiments were conducted on the source code of Firefox version 2.0 (Linux configuration).

No. of macro definitions	No. of macros called	No. of full expansions
33214	15648	305117

Table 6.1: Summary of macro definitions and expansions

The number of macro definitions and full expansions found in the source code are listed in Table 6.1. The number of macro calls is high, there being 90 macro definitions which are called over 1000 times. Table 6.2 contains the number of total calls in the configuration and also the size of slices computed for each macro definition.

	Individual calls	Slice sizes
Average	53	43
Median	2	4
Max	47,046	20,040
Min	1	1
Sum	834,866	674,440

Table 6.2: Summary of macro calls and slice sizes

Slice size has been investigated on procedural programs both for static slicing [11] and for dynamic slicing [9]. In our case the slice size is compared to the size of the graph. We used the number of nodes as the size of the graph, which is the sum of the calls and definitions.

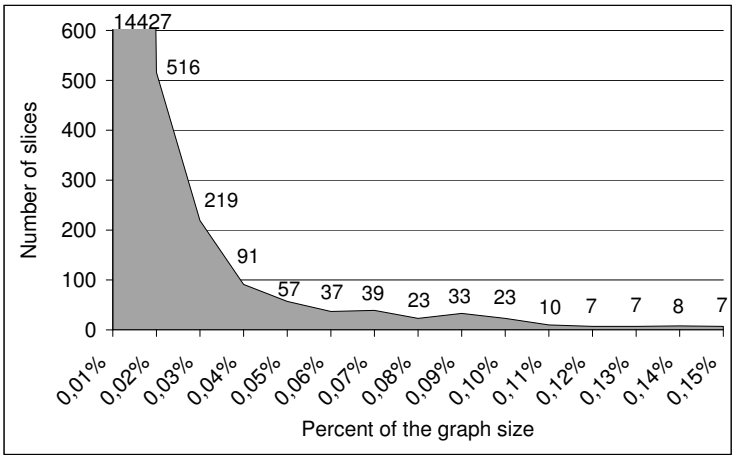

Figure 6.6: Histogram of slice sizes relative to the graph size

Figure 6.6 shows a histogram of the relative macro slice sizes. The shape of the histogram is just as we expected. The majority of the slices are smaller than 0.01% of the graph size. In the figure the area associated with this value has been removed. Also, there are 144 slices which are larger than 0.15%, and which have been omitted from the figure. (Their sizes are between 0.15% and 6.25%.) The sizes are relatively small, which is one advantage of the approach, but they tell us that in many cases it is hopeless to try to locate them by hand.

The use of ## operators to create macro calls is another issue which motivated our research and the development of the tool. The number of definitions containing a call with the concatenate operator is 24. There were 337 macro calls made via these, which confirms that this strange construction does indeed occur in real-life software.

The macro slicer tool required 1GB of RAM, and computing all the slices on the graph took 60 seconds. The preprocessor schema instance is not fine-tuned to macro related analysis; it contains information about other preprocessor constructs too. For macro impact analysis purposes, the graph can be stripped to contain only the necessary details about macros.

6.7 Summary

In response to the lack of a satisfactory solution to the macro change impact problem, we introduced an approach based on macro slices. Using the relations between macro invocations and definitions, we have constructed the Macro Dependence Graph on which macro slices can be computed. We discussed the notions of C/C++ and macro slicing, the defined forward and backward slices By using multiple edges and edge coloring, this graph handles potential (and later defined) macro names and macro redefinitions. Edge coloring, according to the full macro expansion paths, also ensures that the graph is acyclic even when it covers not only one compilation unit but a whole software system. This is important because software maintenance and program comprehension tasks many times require inter-unit dependencies covering the whole source tree.

A solution for macro slicing is outlined using preprocessor schema instances as dependence graphs. To show that this concept is sound, an experimental tool based on the Columbus C/C++ frontend [38] was developed. We then carried out experiments which proved the validity of our approach, and we showed that the tool is able to create slices for large software (e.g. Mozilla Firefox). In the same time, we have given an insight to the distribution of slices in a concrete case.

Main results of this chapter belong to contribution point II/1 (Macro slicing) and partly to point II/3 (Experimental evaluation of slicing methods), and were published in paper [VBF07].

7 Combining preprocessor and C/C++ language slicing

7.1 Introduction

In the previous chapter the notion of *macro slicing* was introduced. The method is appropriate for revealing macro-related dependencies, but with this method the slices are computed on preprocessor constructs only hence the slices are restricted to macro constructs.

Having seen the weaknesses in this respect of existing C/C++ language slicers today, it seemed promising to *combine* macro slices and regular C/C++ language slices computed based on the dependencies between the syntactic elements of the source code (referred to as *language or C/C++ slices* is the following). The combined slice contains more accurate information about the C/C++ program as we will see later on, which is very important from a program comprehension point of view. In the combining process a special role is played by the toplevel macro calls (which are in the program text, not in the macro definition text). A toplevel macro call is a part of macro slices, but when it is fully replaced, the resulting text is part of the C/C++ program, therefore it may be a part of the C/C++ slices as well. This way the endpoint of a macro slice serves as a starting point for a language slice. We can also define a similar

combination in the opposite direction, in which case a C/C++ slice may be a starting point for preprocessor slices. At this point, the reader should again see the motivating example given in Section 5.2.1.

In the next section we will discuss in detail the connection between the two kinds of slices. Implemented tools and algorithms used are described in Section 7.3, while Section 7.6 reports the empirical results of our implementation. In the last section we round off with some conclusions and ideas for future study.

7.2 Combining C/C++ language and macro slices

The process of combining the two types of slices can be performed in both the forward and backward directions. In the forward direction the slicing criterion is a macro definition. The macro slice contains toplevel macro calls as connection points, the replaced toplevel macro calls being (part of) C/C++ program elements whose program elements serve as slicing criteria for regular language slicing. The final slice contains both preprocessor and C/C++ program elements. The backward direction is similar, but here the slicing criterion is a C/C++ program element, and the language slice may contain program elements which are in turn parts of the result of a macro call. These macro calls are used for macro slicing and the final slice contains the language slice and all the macro slices as well.

Combining macro and language slices requires that a common set of nodes and edges be defined with the dependency relation as well. C/C++ language slices are usually computed on some kind of a Program Dependence Graph (PDG) [81], or more generally on a System Dependence Graph (SDG) introduced by Horwitz *et al* [55]. The PDG models interprocedural dependencies between procedures where each procedure is modelled with a PDG. In the following we shall consider a generalized SDG, on which a general C/C++ dependency relation is defined (called dep_{cc}). The terms and definitions used in macro slicing are given in sections 6.2 and 6.3. To fully understand this section the reader should read those parts on toplevel macro invocations, the macro dependency relation, the Macro Dependence Graph (MDG) and macro slices. In the preprocessor case, the MDG can be constructed in such a way that it contains dependencies from every compilation unit in a program; there is no need to define two kinds of graphs for the macros.

The MDG can be used in combination with the SDG in the following way. Both of

them have a well-defined structure, the only problematic point being the connection. The MDG is based on the original source code, while the SDG contains C/C++ language elements. In practice it is based on the preprocessed code (`.i` file). The toplevel macro invocation (call) serves as a connection point (see the motivating example in Section 5.2.1). From the point where the macro call is replaced with the replacement text, the source code is in C/C++ language form and consists of C/C++ program elements.

Unfortunately, there is no guarantee that the replacement text will be a C/C++ syntactical unit. Moreover, the SDG is composed of program elements, but contains various kinds of nodes like declaration, expression, return and so on. There is a many-to-many relation between macro replacement texts and SDG nodes. For instance the macro replacement may be a sequence of statements that is represented by several nodes in the SDG, and the macro may even be a constant that is only a part of an SDG node. An SDG node, which at least partly comes from a macro replacement, depends on the macro itself. Thus a dependency relation can be defined based on shared characters between the SDG node and the macro (replacement). Let $repl(a)$ be the replacement text after a full expansion of macro call a, where $repl(a)$ consists of characters with their position in the preprocessed file. (The SDG node b also contains characters with their position in the preprocessed file.)

Definition 12 *Let $dep_{comb} \subseteq MDG \times SDG$, $a \in MDG$, $b \in SDG$, $b \in dep_{comb}(a)$ if and only if a is toplevel and $\exists x$ character: $x \in repl(a)$ and x is contained by b.*

An SDG node depends on an MDG node if at least one of its characters comes from the replacement of the MDG node.

Using the definitions given in this section, the combined slice can be defined. The dep_{cc} C/C++ dependency relation, the dep_m macro dependency and the dep_{comb} combining dependency relations have already been presented. Next, let DG be the combined dependence graph and dep the combined dependency relation:

Definition 13 *Let*

$$DG = SDG \cup MDG$$

and

$$dep(x) \subseteq DG \times DG = \begin{cases} dep_m(x) \cup dep_{comb}(x), \textit{if } x \in MDG \\ {dep_{cc}}^{-1}(x), \textit{if } x \in SDG \end{cases}$$

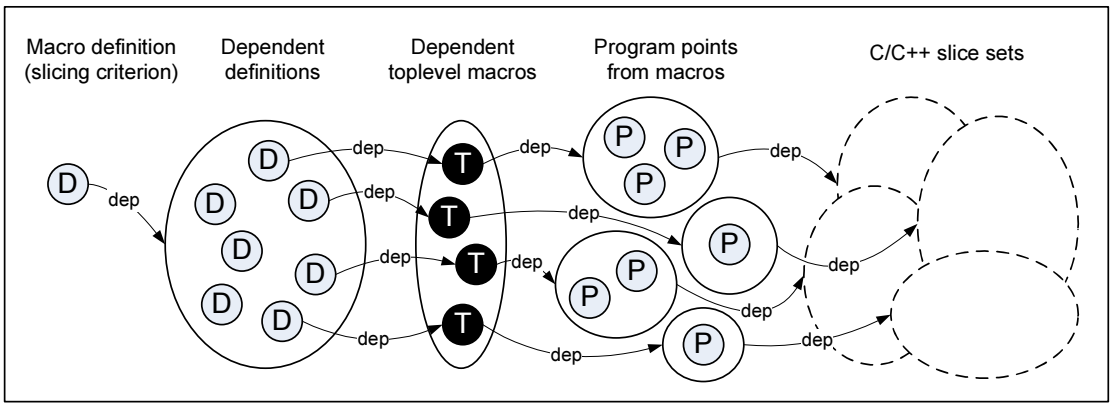

Figure 7.1: The forward direction for combining the slices, with the dependency relation between macros and C/C++ program points

Note that the dep relation uses the inverse of the dep_{cc} relation. In program slicing the direction of the dependency relations usually points in a backward direction. However, in the case of macro slicing the direction is the opposite of the macro call relation. To be consistent, for combined slicing the inverse of the C/C++ dependency should be used.

Definition 14 *Let $< p, x >$ be a slicing criterion, where x is a variable at program point p. Let $k \in DG$ be the corresponding graph element for x. The combined forward slice of the criterion is the set of program points, which corresponds to the $\{l \in DG \,|l \in dep^t(k)\}$ set, where dep^t is the transitive closure of the dep relation.*

Definition 15 *Let $< p, x >$ be a slicing criterion, where x is a variable at program point p. Let $k \in DG$ be the corresponding graph element for x. The combined backward slice of the criterion is the set of program points, which corresponds to the $\{l \in DG \,|k \in dep^t(l)\}$ set, where dep^t is the transitive closure of the dep relation.*

The forward direction case is depicted in Figure 7.1. The capital letters in the figure elements refer to their type and not their name. The slice starts at the slicing criterion, which is a macro definition (D). There is a set of dependent definitions (D), and there is a set of dependent toplevel macro invocations (T). (Note that many dependency edges among the elements of this set have been omitted here.) When toplevel invocations are replaced, the result of each invocation becomes a part of C/C++ program elements (P). A regular language slicing algorithm computes the slice for each program element,

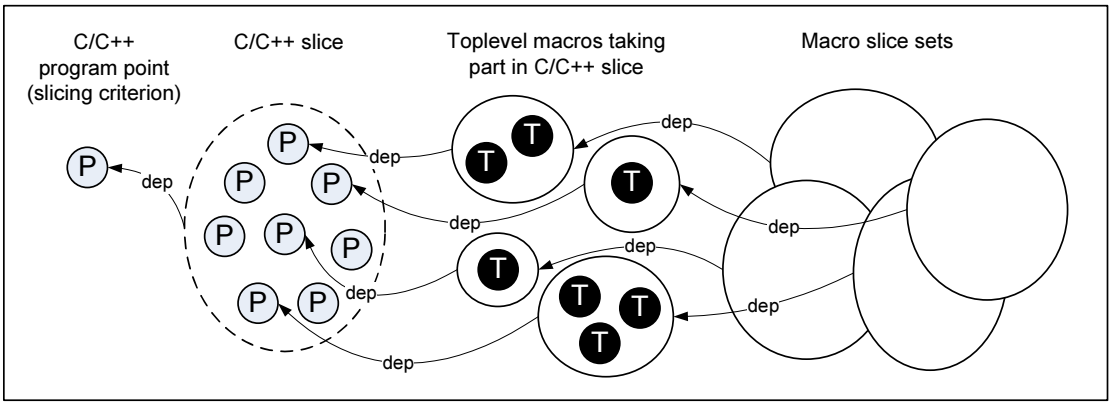

Figure 7.2: The backward direction for combining the slices, with the dependency relation between macros and C/C++ program points

hence the final combined slice contains every element in the figure.

The backward direction case is depicted in Figure 7.2. Here once again the capital letters in the figure elements refer to their type and not their name. The slicing criterion is a C/C++ program element (P). The slice may contain SDG nodes which are (at least partly) the results of one or more macro invocations. The toplevel invocations which are present in the C/C++ slice can be found along the dependency edges. For each of these toplevel invocations, macro slice sets can be obtained using backward macro slicing. The final combined backward slice contains all the same elements as those shown in the figure.

The combined graph and the combined slices of the sample source code from Section 5.2.1 can be seen in Figure 7.3. Nodes corresponding to forward and backward slices are denoted by a capital 'F' and 'B', respectively. The toplevel macro call `DECLI(i,2)` is present in both of its possible forms: as a macro and as a program point. The forward slice contains every node except the definition of `SGN`, while the backward slice contains every node of the graph.

Note that the method does not make use of any special information concerning the SDG of the C/C++ slicing algorithm. Just the dependency relation and the character positions of the node texts are used. Hence, in theory the method can be used for static or dynamic slicing. Moreover, it does not matter whether data, control or some other dependency relation is used for slicing.

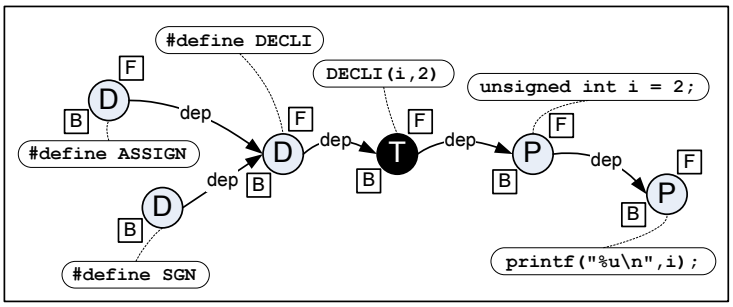

Figure 7.3: Nodes and slices of the motivating example

7.3 Tools

The formal definitions of combined slices were given in the previous section. A combined slicer can be implemented in various ways. There are three tools that must be used to implement the method: a macro slicer, a C/C++ slicer, and a combiner tool which implements the connection between them. In this section we will describe our implementation. In our experiments, the slices were computed for each appropriate node in dependence graphs, so our tools and algorithms are global in this sense. The algorithms given in the following section follow the way the tools work. To create an on-demand version of the toolchain – which computes slices only for criteria given as input – minor changes are required (overviewed below).

In our toolchain the macro slicer is built on top of the CANPP tool as mentioned in Section 6.5. The macro slicer tool analyzes the project and afterwards creates a graph instance of the Columbus preprocessing schema (Figure 3.2). The graph contains dependency edges between preprocessor elements, therefore it can be used as an MDG on which macro slicing can be performed. For the C/C++ part we implemented a CodeSurfer plugin to get slicing information [49]. Similar to the previous case, CodeSurfer builds the SDG graph representation from a software project and determines language level dependencies (and other pieces of information as well). CodeSurfer provides access to the internal representation of the SDG and the dependency information via plugins. We used the C API, which just offers core functionality, but it is suitable for slicing (the Scheme API provides full access).

A logical outline of the toolchain is depicted in Figure 7.4. The toolchain consists of the core analyzers (Columbus and CodeSurfer), the macro slicer tool, the CodeSurfer slicer plugin and a small combiner tool which summarizes the results obtained (the

combiner is implemented together with the macro slicer). The tools communicate with each other via a set of toplevel macros (given by their line information), which is the common point of the two slicers.

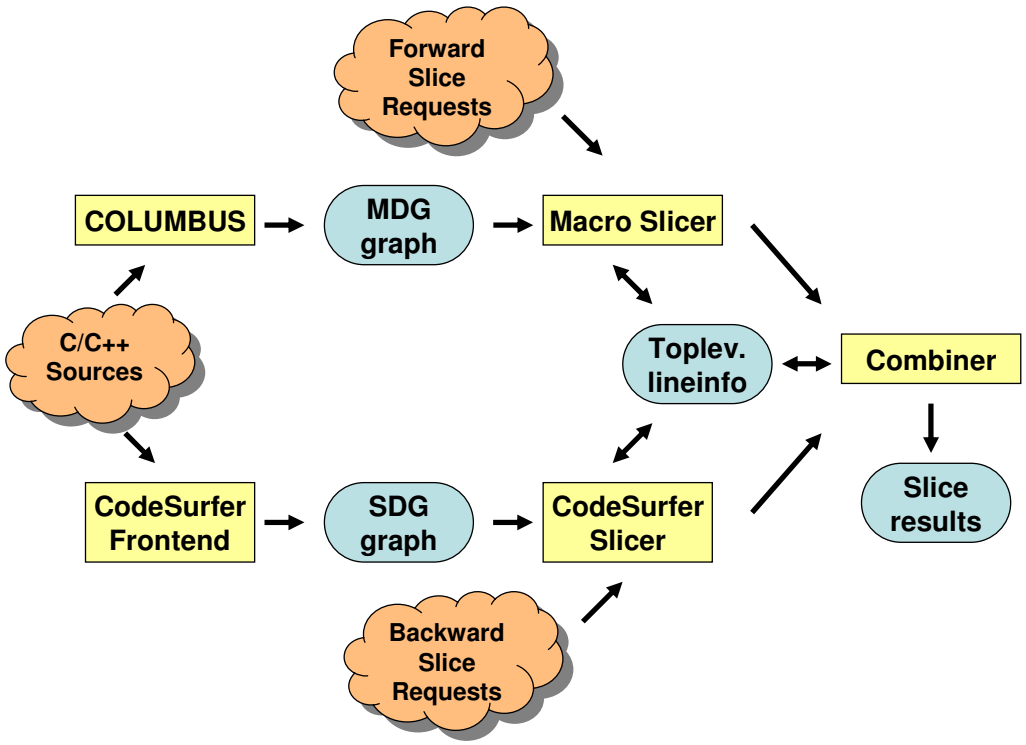

Figure 7.4: Logical tool architecture - forward and backward slicing

The slicing process in both the forward and backward cases starts with the core analyzers. In the backward direction the C/C++ slices are continued with backward macro slices at points of macro calls. The CodeSurfer plugin produces the backward slice based on the criterion (which is a C/C++ program point). The slice is then scanned for vertices which are results of toplevel macro calls (matching). The slice details are written into the output, and the set of toplevel macros present in the slice is given to the macro slicer tool. The macro slicer computes backward slice sets for each toplevel macro given as input, and by doing so extends the existing slice. Lastly, the slices are summarized.

In the case of forward slicing the slicing criterion is a macro definition. The macro slicer produces the macro slice of the criterion, whose final result contains the set of toplevel macros, which is then given to the language slicer. In the next part the CodeSurfer plugin identifies positions in the source code where macro replacement was

performed and the toplevel macros are matched with C/C++ vertices. The matching between toplevel macros and vertices is carried out based on line and column information (from the various types of vertices, just those which have a position in the source are used). Next, the language slicing algorithm is executed to produce slices for each vertex, which is then matched with toplevel macros. The results are summarized for each starting macro definition criterion (the C/C++ part of the final slice is the union of the C/C++ slices belonging to the toplevel macros).

7.4 Algorithms

In this section we will provide details about the implemented algorithms. The logical architecture, shown in the previous section, has been slightly altered: the combiner is implemented inside the macro slicer. Hence two algorithms are used both in the backward and forward case: one for the CodeSurfer plugin and one for the macro slicer and combiner. The CodeSurfer plugin is first run in both cases followed by the macro slicer and combiner. The following notation is used in the algorithm descriptions: the Cs and M prefixes refer to CodeSurfer (C/C++) and Macro artifacts, respectively; $vertex$ means a node in the SDG, while $toplev$ means a toplevel macro invocation.

7.4.1 Backward algorithm

Our combined backward slicing algorithm is given in Figure 7.5. As mentioned before, the backward direction means that the C/C++ slices are continued with backward macro slices at points of toplevel macro calls. The plugin gets the vertices from each procedure and then computes the backward C/C++ slice on the project SDG (the function $GetProcedureVertices(SDG)$ returns vertices contained in procedures which have source file positions). Each such slice is scanned one vertex at a time, and the set of matching toplevel macros is found. The $Match(y,AllToplevs)$ function returns the matching toplevel macro set for a vertex ($AllToplevs$ denotes the set of all toplevel macros; for matching, see Section 7.5). The toplevel macros are combined for each such C/C++ slice. The triplet with the original vertex, the associated C/C++ slice and the set of toplevel macros are computed for each criterion and the result is passed to the macro slicer and combiner.

In the second step the macro slicer and combiner produces the final slices for each vertex passed as input. First, the C/C++ slice is part of the final slice. Second, the

set of included toplevel macros is used to compute additional backward macro slices. These macro slices are then placed in the final slice set. The result is the combined backward slice.

In the backward direction the toolchain may work in an on-demand way; in this case the plugin in line 2 of the algorithm iterates through the vertex set passed as an argument.

7.4.2 Forward algorithm

In the forward direction the slicing criterion is a macro definition. The forward macro slices are combined with C/C++ slices via toplevel macros matched with SDG vertices. In this direction the toolchain acts as a global slicer. The CodeSurfer plugin prepares toplevel macros and the associated C/C++ slices for the whole program. The prepared data is passed to the macro slicer and combiner, which computes macro slices and creates the final sets.

Figure 7.6 shows details of the combined forward slicing algorithm employed. The CodeSurfer plugin iterates through all vertices inside procedures and tries to find matching toplevel macros. In the case of a successful match, the forward C/C++ slice of the current vertex is computed, and the set of matched toplevels is paired with the C/C++ slice. The output of the plugin is the set of toplevels paired with the forward slices starting from the matched vertices. The macro slicer and combiner iterates through all macro definitions in the MDG (with the help of the $GetDefinitions()$ function). For each definition the forward macro slice is computed, which will form part of the final combined slice. The macro slice contains (usually several) toplevel macros (provided by the $GetToplevs()$ function). For each included toplevel macro in the macro slice ($GetToplevs()$ function), the set of C/C++ slices is obtained from the input ($GetCsFwSlice()$ function) and then added to the combined slice. The final result is a set of combined slices paired with the associated definition.

Creating an on-demand slicer requires that the macro slicer and the combiner be separated and the tools be called in the following order: macro slicer (with input criteria), plugin, combiner.

CodeSurfer plugin - Backward slice
input: SDG : SDG of the analyzed project
output: $outS$: set of $< v\,, CsBwSlice_v\,, T_v >$ triplets where:
v : vertex $\in SDG$
$CsBwSlice_v$: backward C/C++ slice of v
T_v : set of toplevel macros in $CsBwSlice_v$
begin
1 $outS = \emptyset$
2 **foreach** $v \in GetProcedureVertices(SDG)$
3 $T_v = \emptyset$
4 $CsBwSlice_v$ = compute backward C/C++ slice for v on SDG
5 **foreach** $y \in CSBwSlice_v$
6 $T_v = T_v \cup Match(y\ ,\ AllToplevs)$
7 $outS = outS \cup < v, CsBwSlice_v\,, T_v >$
end

MacroSlicer & Combiner - Backward slice
input: MDG : MDG of the analyzed project
inS : set of $< v\,, CsBackSlice_v\,, T_v >$ triplets
output: S : set of $< v\,, S_v >$: pairs - combined slice set
for each request (vertex)
begin
1 $S = \emptyset$
2 **foreach** $< v\,, CsBackSlice_v\,, T_v > \in inS$
3 $S_v = CsBackSlice_v$
4 **foreach** $x \in T_v$
5 $MBwSlice_x$ = compute backward macro slice for x on MDG
6 $S_v = S_v \cup MBwSlice_x$
7 $S = S \cup < v\,, S_v >$
end

Figure 7.5: Computing combined backward slices

CodeSurfer plugin - Forward slice
input: SDG : SDG of the analyzed project
output: $outS$: set of $< T , CsFwSlice_T >$ pairs where:
T : set of toplevel macros
$CsFwSlice_T$: forward C/C++ slice connected to T
begin
1 $outS = \emptyset$
2 **foreach** $v \in GetProcedureVertices(SDG)$
3 **if** $Match(v, AllToplevs) \neq \emptyset$
4 $CsFwSlice_v$ = compute forward C/C++ slice for v on SDG
5 $outS = outS \cup < Match(v, AllToplevs) , CsFwSlice_v >$
end

MacroSlicer & Combiner - Forward slice
input: MDG : MDG of the analyzed project
inS : set of $< T , CsFwSlice_T >$ pairs
output: S : set of $< d , S_d >$: pairs - combined slice set
for each request (macro definition)
begin
1 $S = \emptyset$
2 **foreach** $d \in GetDefinitions(MDG)$
3 $MFwSlice_d$ = compute forward macro slice for d on MDG
4 $S_d = MFwSlice_d$
5 **foreach** $t \in GetToplevs(MFwSlice_d)$
6 $S_d = S_d \cup GetCsFwSlice(inS, t)$
7 $S = S \cup < d , S_d >$
end

Figure 7.6: The combined forward slicing algorithm

7.5 Details on matching and graph coloring

There are many factors which make the matching of macros and vertices based on file position a challenging task. The behaviour of the tools had to be adjusted in many

areas including the physical and logical lines (e.g. for the `#line` directive CodeSurfer preserves the original line information), handling macros in conditional directives, and handling macros defined in the command line. The plugin iterates through vertices belonging to procedures, which means that some vertices are omitted such as forward declarations or globals). Another important factor is the handling of standard libraries. The SDG contains additional vertices from standard libraries, and some vertices used in its internal representation. Accordingly, the macro slicing tool is adjusted to match macros from standard libraries, but not to report errors for omitted ones.

The matching process is based on comparing source position intervals. The result of the $Match$ ($vertex$: y , $set < toplev >$: T) function is the subset of T. The $repl(a)$ function supplies the replacement text after a full expansion of macro call a, where $repl(a)$ consists of characters with their position in the preprocessed file. If the vertex y contains characters from $repl(m)$ (i.e. the expansion of the toplevel macro $m \in T$), then the matching set contains m. In other words, the matching algorithm checks the file position of the vertex and the replacement of macros, and if there are overlapping intervals then the matching is successful.

A schematic view of the matching process is shown in Figure 7.7. The toplevel macro (T) is expanded using two definitions (D_1, D_2). The final replacement is denoted by $repl(T)$ in the figure. The result is included in the C/C++ analysis, and the SDG vertices are defined based on the preprocessed source including $repl(T)$. Vertices are denoted by horizontal lines as they may cover the same source position.

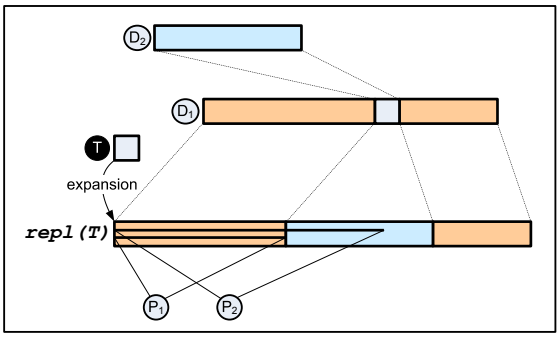

Figure 7.7: Matching based on common characters in an expansion

The figure contains two successful matches; namely, (T) is matched with both (P_1, P_2). Note that the replacement text is included in matching in full. Lastly, the combined forward slice requested on D_2 consists of the set $\{(D_2, D_1), T, (P_1, P_2)\}$.

The matching algorithm can be refined with a more accurate check on positions. If we track the origin of the fragments contained in the replacement text, then the slice set may be smaller. In this case P_2 is matched with fragments from both D_1 and D_2, but P_1 only matches fragments from D_1. Therefore using accurate tracking the combined forward slice on D_2 does not contain P_1; it consists of $\{(D_2, D_1), T, (P_2)\}$. This kind of slicing produces smaller, more accurate slices. Despite this, the result is not always better (here it is not obvious that P_1 is not related to D_2). Another question arises about the interpretation: should D_1 contained in the forward slice of D_2? The toolchain in our experiments applied the first type of matching process without tracking macro fragments.

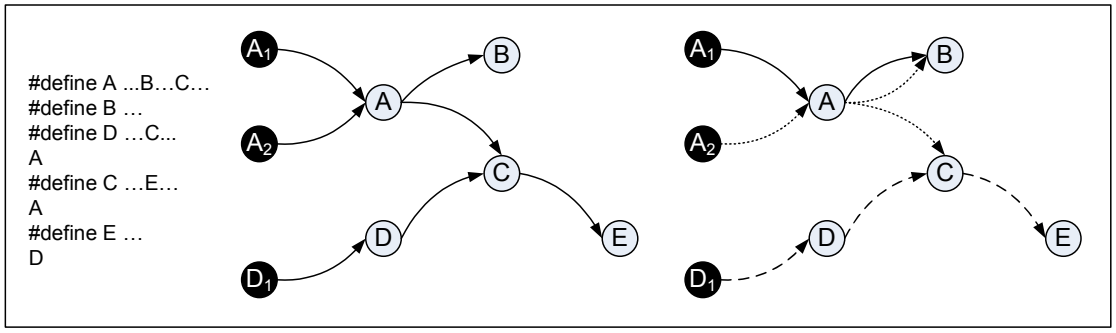

Figure 7.8: Edge coloring example

The graph coloring method was outlined in Section 7.2. An illustrative example for it is given in Figure 7.8. The program code is given on the left hand side of the figure, which is followed by the basic and the colored version of the graph (colors are represented by solid, dotted and dashed lines). Coloring reflects the full macro expansion of toplevel macros. For instance macro call A_1 uses the definition of A (solid lines). During the further expansion macro B is also expanded, but C is not defined at that source position. Using the basic dependence graph for macro slicing would result in inaccurate (larger) slices. Computing forward macro slice on the definition of E, the basic graph would result in the set $E, C, D, D_1, A, A_1, A_2$. Using the dashed edges in the colored graph a much better slice can be computed, namely E, C, D, D_1. Coloring helps in a similar way in the case of backward macro slicing. The backward macro slice computed on A_1 using the basic graph includes unnecessary nodes (C, E). Although the example given is rather artificial, the analyzed projects contain several complex preprocessor constructs, which confirms the necessity of graph coloring.

7.6 Measurements

7.6.1 Subject programs

Experiments are performed on 28 open source projects, starting from small programs to medium size ones with about 20k lines of code. Many of the programs are selected based on comprehensive empirical studies on slicing [11] and preprocessor usage [25]. We found a total of 240k non-empty lines of code enough to prove the usability of the method. Table 7.1 contains a list of projects used in our measurements and their basic statistics. Sizes are given in non-empty lines of code as CodeSurfer calculates its LCode metric (note that this metric is significantly smaller than the usual LOC metric, when usually comments and empty lines are counted). The build time of dependence graphs is given is seconds, as the *time* unix tool reports the user time of the process. The building time includes the time needed to build the project, not only the graph building phase. The number of nodes in the graphs can be used as a measure of the graph size. Not surprisingly, the MDG is smaller than the SDG, which is almost 60 times larger on average. The time required for slicing operation is given in the tables, backward and forward slicing being done during the same run. The memory consumption was below 350M for the CodeSurfer plugin and below 2.5G for the macro slicer and combiner tool (without any special effort made on decreasing memory consumption).

7.6.2 Slices in detail

In our experiments the measure for the slice size was the number of source code lines which contain vertices from the slice, since this seems to be the best common denominator for different slicing tools. Other researchers have also used this approach [11].

Because of the difficulties in matching, which were outlined in the previous section, there were slices in both directions which the tools failed to match. The failure rate was generally about 8% in the forward case, and under 1% in the backward case, which we found acceptable for reporting measured data. The data given in this section just contains the perfectly matched slices.

The number of combined forward slices and their average sizes can be seen in Table 7.2. We computed all possible forward slices, meaning that we started from each macro definition, and measured the sizes of the individual macro and language slices along with the combined slices. The numbers listed are the average slice size values. We treated the set of toplevel macros in a special way: we added toplevel

Program name	Size (LCode)	MDG build time (s)	MDG size (nodes)	SDG build time (s)	SDG size (nodes)	Macro slicing time (s)	C/C++ slicing time (s)
replace	512	0.28	136	1.18	3205	0.26	7.85
copia	1085	0.45	7	6.13	94390	0.12	208.65
time	1119	1.88	162	4.15	5633	0.26	3.73
which	1246	1.87	146	5.41	7449	0.48	29.44
compress	1335	0.84	108	2.18	4408	0.16	8.29
wdiff	1364	2.12	217	4.57	7640	0.53	10.77
ed	2637	3.80	117	9.98	39412	0.73	716.82
barcode	2807	6.34	381	13.76	27970	3.1	427.62
tile	3549	1.93	1881	27.69	51095	19.72	146.43
acct	4008	9.37	899	12.50	24619	5.0	116.98
li	4793	10.71	1826	3006.31	943340	79.9	56238.38
EPWIC	5249	12.10	852	14.68	27099	12.23	443.48
lightning	5563	20.8	1750	69.42	56778	6954.21	572.75
gzip	5997	9.88	1725	17.88	37525	34.16	1315.92
userv	6016	5.47	1244	24.72	105902	23.30	3281.28
indent	7582	4.55	857	12.22	42102	17.98	1100.14
bc	9472	9.6	1554	24.90	59503	31.17	2080.13
diffutils	10124	18.91	1971	29.35	53928	31.54	1261.76
gnuchess	11045	13.87	2511	29.12	70782	143.8	4391.19
ctags	11670	12.96	1480	55.31	209357	106.61	12611.60
sed	13339	9.37	2527	26.28	89788	204.76	9374.67
nano	13698	14.96	3964	38.11	177879	591.88	23445.10
ijpeg	15253	25.82	4283	39.75	77531	212.62	6948.48
flex	17533	22.56	3188	112.12	126757	259.55	9912.45
bison	20673	35.74	4387	88.64	138972	98.92	16099.25
wget	21104	27.88	4146	95.28	269209	993.85	60294.88
espresso	21780	3.86	0	52.79	151802	0.18	9642.20
go	22118	5.40	5296	22.18	110236	499.19	22550.61
Total	242671	293.32	47615	3846,61	3014311	10326,21	243240,85

Table 7.1: Subject programs

macros to both the macro slice and the associated vertices of the C/C++ slice as they belong to both kinds of slices.

There are two items of especial interest in the list. The program *espresso* is interesting because it does not contain any macro definitions. The program *lightning* is exactly the opposite: it is the only one that has larger macro slices than C/C++ slices. Examining the code confirms that some C source files of this program are full of macro definitions and calls.

Program name	Number of slices	Macro slice size (avg)	C lang slice size (avg)	Combined slice size (avg)	Macro slice % (M/C lang)
replace	23	7.1	328.7	335.9	2.16
copia	2	3.0	1132.5	1135.5	0.26
time	31	8.9	287.6	296.5	3.09
which	22	6.3	544.8	551.1	1.16
compress	26	4.7	277.9	282.6	1.69
wdiff	25	7.0	300.6	307.6	2.33
ed	27	3.3	1459.2	1462.5	0.23
barcode	44	6.7	1665.8	1672.5	0.40
tile	145	23.1	2468.0	2491.0	0.94
acct	76	14.2	761.9	776.1	1.86
li	111	29.7	3966.6	3996.3	0.75
EPWIC	122	7.1	1102.5	1109.6	0.64
lightning	341	983.8	167.5	1151.3	587.34
gzip	259	11.0	3274.0	3285.0	0.34
userv	202	12.2	2732.3	2744.5	0.45
indent	53	16.8	4518.0	4534.8	0.37
bc	153	9.3	3832.7	3842.1	0.24
diffutils	242	13.7	2997.5	3011.2	0.46
gnuchess	242	14.8	7086.4	7101.2	0.21
ctags	111	17.0	8337.1	8354.1	0.20
sed	256	102.6	8812.6	8915.2	1.16
nano	389	18.3	12590.1	12608.4	0.15
ijpeg	322	19.1	5861.3	5880.4	0.33
flex	334	16.3	9036.2	9052.5	0.18
bison	248	28.0	4458.6	4486.6	0.63
wget	340	27.2	16045.4	16072.6	0.17
espresso	0	nan	nan	nan	nan
go	382	38.2	10817.2	10855.4	0.35
Total/avg	4528	$\sim$ 96.62	$\sim$ 6525.98	$\sim$ 6622.60	1.48

Table 7.2: Summary of forward slices

Backward slices may not necessarily contain macro calls. Although the average

number of macro calls is not so high, most of the backward slices contain macro calls (above 75%). The number of computed combined slices (which necessarily contain macros) and their average sizes are given in Table 7.3, where we used the same approach for measurements as we did with the forward slices. It can be seen that backward macro slices are generally bigger than those for forward slices, which can be explained by the fact that language slices usually contain many more code lines, and hence more potential starting points for macro slices exist (we used both data and control dependencies for slicing C code). Another reason might be that in the backward case we produce slices for each vertex, so more of the large slices are counted, while in the forward case we selected just a few vertices (according to the macro calls). This way, the average may be higher in the backward case.

The last column in both tables show the ratio of macro slice size relative to the C language slice size in percentage terms. The table tells us that the individual macro slices are relatively small, but this may be due to the size difference of the SDG and the MDG graphs. For a given slicing criterion the smaller the slice the better, naturally without ignoring any dependency. Macro slices are more accurate in this sense, while still having a relatively small additional percentage value.

There is a wide range of open source software which has been analyzed by Ernst et al [25]. They report the preprocessor directive usage in open source software and find that preprocessor directives make up about 8.4% of the program code on average. It is worth mentioning that in both directions the extra code lines coming from macro slices are relatively small compared to the language slices, so their true worth is debatable here. However, we think that in many cases these additions may be crucial from a program comprehension point of view. The real world example given in Section 5.2.2 provides an instance where the macro slice part is small but useful. This is not a rare occurrence. This example was taken from the *flex* program where macro usage is close to the average according to the empirical study mentioned above. In this respect, traditional C/C++ language slices without macro slices can be treated as *unsafe*, overlooking important information about macros.

7.7 Conclusions and future work

The work presented was motivated by the observation that virtually all available program slicing tools for the C/C++ language lack the proper and complete handling

of preprocessor constructs. From a program comprehension point of view, the existing methods often seem inadequate. For instance, the impact of changing a macro definition cannot be accurately followed throughout the program's preprocessor and non-preprocessor related parts. Existing tools either compute the slices based on dependencies in the language constructs or provide rich features to model macro usage, but not both. This could have a detrimental impact on various fields related to program comprehension and maintenance in general. For example, in change impact analysis, a failure to identify a dependency of a change could have the effect of inaccurately predicting the cost of changes and of performing incomplete change propagation, which in turn would result in an increased risk of regression [86].

With our work we sought to fill this gap and proposed a *combined* approach for computing slices in C/C++ programs. We have given the necessary definitions for the combined dependence graph (macro dependence graph is combined with static system dependence graph). Moreover, definitions of forward and backward combined slices are provided, together with algorithms for computing them. We justified our approach by providing a realistic sample program comprehension problem and other possible applications of the method. Existing and newly developed tools were employed in an experimental tool setup with which a number of program slices were computed. We counted the program points returned via the combined approach and compared it to slices without the preprocessor components. The first results measured on open source projects look promising, and clearly demonstrate the benefits of using our approach. We recommend that similar combined strategies for slice calculation in existing tools like CodeSurfer be integrated. In it, one may be able to use and extend the existing internal representation for this purpose.

The results of this chapter belong to contribution point II/2 (Combining C/C++ language and preprocessor slicing) and partly to point II/3 (Experimental evaluation of slicing methods), and were published in research papers [VJBG08, VBG09].

Program Name	Number of slices	C lang slice size (avg)	Macro slice size (avg)	Combined slice size (avg)	Macro slice % (M/C lang)
replace	647	205.2	86.6	291.8	42.20
copia	3044	924.2	6.0	930.2	0.65
time	598	115.4	19.0	134.4	16.46
which	1288	396.1	53.6	449.8	13.53
compress	601	323.7	64.5	388.2	19.93
wdiff	989	170.2	45.4	215.6	26.67
ed	3849	1543.8	37.5	1581.3	2.43
barcode	4143	1569.3	153.1	1722.3	9.76
tile	2469	358.9	193.3	552.2	53.86
acct	3896	492.1	93.8	585.9	19.06
li	7695	4025.3	1392.2	5417.5	34.59
EPWIC	6434	897.0	239.7	1136.7	26.72
lightning	808	101.0	73.0	174.0	72.28
gzip	4701	2884.3	979.7	3864.1	33.97
userv	8002	2163.5	482.7	2646.2	22.31
indent	5781	3196.4	427.3	3623.7	13.37
bc	8548	3108.0	612.7	3720.6	19.71
diffutils	9614	1830.4	397.4	2227.8	21.71
gnuchess	10919	5137.0	1838.9	6975.8	35.80
ctags	12775	7496.0	976.1	8472.0	13.02
sed	12295	7367.3	1571.9	8939.2	21.34
nano	14754	12750.5	3354.6	16105.1	26.31
ijpeg	13479	5661.1	1859.5	7520.6	32.85
flex	12530	7250.8	1977.1	9227.9	27.27
bison	3873	5763.4	1792.0	7555.4	31.09
wget	21471	14345.1	2838.9	17184.0	19.79
espresso	0	nan	nan	nan	nan
go	23840	10304.6	3628.1	13932.7	35.21
Total/avg	199043	$\sim$ 6647.69	$\sim$ 1706.99	$\sim$ 8354.68	25.68

Table 7.3: Summary of backward slices

8 Related work

In the past two decades preprocessing has been a recurring theme in the literature and this has led to the creation of several useful tools. This is still the case today, but there is still a lack of general solutions in many branches of software engineering where the preprocessor is present. First, we shall introduce related work, and then discuss more specific topics, emphasizing some notable solutions. Usable tools and high quality research articles usually go hand in hand. When a research paper is cited, most likely there exists a corresponding tool implementation and vice versa.

8.1 Preprocessor-related problems and solutions in general

In spite of their disadvantages, preprocessor directives are still widely employed. Ernst, Badros and Notkin [25] analyzed the frequency and nature of preprocessor use. In a study they analyzed 26 commonly used Unix software packages written in C with about 970,000 source code lines altogether (for example gcc, bash, emacs, gs and cvs). Among other things they found that preprocessor directives make up the relatively high 8.4% of lines on average (varying from 4.5% to 22%). Deep include hierarchies are necessary properties even for medium-sized programs. Vo and Chen implemented a tool

to analyze include hierarchies [117]. They found that real-life software often contains unnecessary includes. The tool is able to present the include dependencies in both textual form and graphical form, and provides help in rearranging the include hierarchy. At the same time, Grass and Chen presented a C++ information abstractor tool [50], and it also contains a program for the analysis and display of include relationships. A recent paper by Spinellis proposes a solution for the automatic removal of unnecessary includes [102], based on computed dependencies of program elements.

Studies have been done that handle the preprocessing problem in a more general way. Favre studied the role of the preprocessor from a reverse engineering point of view, and listed the main drawbacks of using the preprocessor. He also noticed that the presence of preprocessor directives in programs impose serious limitation on the applicability of worthwhile techniques, especially those which are based on program source transformations [28]. The proposed solution, the APP (Abstract PreProcessor), is an abstract language that handles preprocessor directives in a similar way that other programming languages do [29].

Badros and Notkin [6] constructed a framework, called PCp^3, which executes user defined Perl callback functions when an action of interest occurs during the preprocessing and parsing (after preprocessing they build an AST to handle some C language-level constructs like call graphs). To do the preprocessing part, the authors modified and embedded the GNU C preprocessor library. As an example they presented functions to describe macro expansions and also generated Emacs Lisp source to visualize them. With the help of hooks the conditionally excluded lines can be analyzed as well. The solutions they presented are quite flexible, but the running time may be prohibitive in the case of large software; moreover one has to write custom code for each kind of use and this requires a good knowledge of the details of the tool's implementation.

Kullbach and Riediger worked along similar lines as our team [63]. They divided the code into foldable and non-foldable segments, which are visualized in the GUPRO [24] source code browser. Applying this tool the important parts of the code can be seen more clearly because the programmer can hide and show (fold/unfold) the segments. The fold/unfold structure is used to describe the preprocessor transformations (folded/unfolded state means the code before/after a preprocessor action). The structure is good for visualization and the user can also define custom folds. Since each transformation (macro calls, conditionals, etc.) is described using the same structure, this may be inconvenient for some other preprocessor-related applications. Yet another

difference to our work is that GUPRO deals only with just one configuration.

As part of the Ghinsu program slicing tool, Livadas and Small developed a special preprocessor [68]. They identified five mappings between the original and the preprocessed code. Their preprocessor inserts special lines into the preprocessed file to support Ghinsu's source code highlighting methods. Mappings for macro definitions and invocations are elaborated on in their paper, but that of conditionally excluded code (i. e. configuration independence) was not investigated. Unfortunately, it appears that this project has been discontinued, and from the latest information gleaned we found that the implementation had certain drawbacks (complex projects consisting of multiple source files were not handled), which prohibits its use in real-life programs.

The area of software configurations has also been studied. Spencer and Collyer [99] investigated the use of conditional directives for separating codes running on different platforms in the early years. Their opinion is that the wide use of conditionals is "harmful" and should be avoided where possible. Well-organized code should be used instead. Krone and Snelting [62, 96] analyzed the complex configuration structures created with directives and produced a graphical output of them. Concept lattices were used to help in reengineering configurations [97].

Latendresse [64] proposed a solution for finding the conditions required for a particular source line to get through the conditional compilation. The approach uses so-called conditional values to represent conditional directives. Further details including rewrite systems were published in [65]. The advantage of this approach is that the efficient symbolic evaluation algorithm has linear time complexity. CViMe (Conditional-compilation Viewer and Miner) and its continuation the C-CLR tools are Eclipse plugins, which collect configurations controller macros and then provide source code views on user selected configurations [94, 93]. A similar functionality is provided by the Sunifdef command line tool [106]. It takes defined macro symbols as arguments and attempts to eliminate or simplify conditional directives.

Baxter and Mehlich introduced a method for removing unnecessary conditional directives based on rewrite rules of the DMS system [7, 54]. Aversano et al. offered a solution for preprocessor-conditioned declarations, which occur when C variable declarations depend on preprocessor conditions [5]. This construction may be harmful especially in case of the presence of rare, un-maintained configurations, which may lead to type confusions of variable declarations and usages.

Sutton and Maletic implemented analyzer tools on the top of the srcML infras-

tructure to reveal portability issues based on include files and configuration macros [107]. Observations from the analysis of three mainstream software libraries were used to draw conclusions about portability best practices. Somé and Lethbridge proposed a heuristic solution for selecting most important, relevant configurations, assuming that analyzing all configurations is impossible, but performing the analysis for the most important configurations gives significantly better results than one-configuration parsing [98]. Rieger et al. reported a study on teaching the FAMIX system to handle preprocessor directives [69]. The Famix metamodel is extended by notions representing properties of most common directives including conditional compilation and macro usage.

In the work of Garrido the analysis of preprocessor constructs was integrated into the C refactoring tool [40]. Garrido tackled the problem of refactoring directives [41] and implemented a configuration independent solution [42, 43].

Vittek also tackled problems in refactoring related to the preprocessor [116]. His Refactoring Browser [114, 115] carries out automated modifications on a C source code. An interesting idea in this paper is that of handling macros as special include-files (the macro body is "included"), but the handling of ## operators is not solved in some cases. To handle the problem of configurations, this tool relies on user input.

Cox and Clarke proposed an XML annotating technique for maintaining the mapping between the original and the preprocessed forms of the source code [20]. Another interesting idea was presented for mapping unprocessed and preprocessed code by Gondow et al. [48]. The TBCppA tool instruments the original source code and marks macro definitions and calls, so the mapping can be recovered after using a native preprocessor.

8.2 Refactoring

A lot of effort has been made by researchers to provide a formalism for refactorings. In Chapter 4 we mentioned some articles on refactoring, along with some of the available tools. The graph transformation approach to formal refactoring is a well-known method. A remarkable summary of this topic can be found in the papers by Mens et al. [72, 73]. 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. [15] use a similar formalism, the focus being on the coordination of a

change in different model views of the code using distributed graph transformations. These contributions seek 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 [27] provided a natural way of implementing refactorings. The paper shows the key points, but this solution somehow lacks the formal basis. They report that these 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 aspect. Those working on C or C++ analyzers are confronted by the problem of preprocessor directives. Therefore, a lot of effort has been made to avoid their usage. Mennie and Clarke proposed a method to transform some macros and conditionals into C/C++ code [70]. The authors classified the different uses, removed some preprocessor constructs, and also discussed the pitfalls of the method. Spinellis tackled the problem of global renaming of variables [100]. The identifier tokens are classified and only the right occurrences are modified. Preprocessor-aware solutions have been implemented in the CScout tool [101].

The preprocessor-problem occurs also in the context of aspect mining and aspect-refactoring. Adams et al. worked on the problem of aspect refactoring, and also how to refactor various conditional compilation usage patterns into aspects [2, 1].

The process of refactoring is strongly affected by the above problem, as implementing a refactoring just on a C/C++ AST may lead to errors. The usual approach is to work on preprocessed code, or to recognize (partially handle) directives [111]. Handling directives is easier when preprocessor constructs form complete syntactical units. Vittek [114] introduced a tool which implements some preprocessor-safe refactorings on C++, but he acknowledged that there are unhandled cases caused by complex code constructions of the two languages (see [120] as well).

A recent work of Garrido also incorporates directive usage into the C language. The presented C refactoring tool includes configuration independent solutions [42, 43]. Several preprocessor-related refactorings can be found in [41], which are 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. Refactorings made on directives are slightly less complicated than language dependent refactorings combined with directives [40], but they can also have an important role in the refactoring of real-life C/C++ programs.

8.3 Slicing

There are relatively few slicing tools available for C/C++ programs. Binkley and Harman [11] conducted an empirical study of the static slice size of C programs and they mention three general purpose slicing tools: Unravel [112], Sprite [75] and CodeSurfer [49], using the latter in their experiments. Unravel was a research prototype that was developed in a discontinued project. It has a number of deficiencies, including the fact that it can only accept preprocessed ANSI C code, which makes it clear that handling macros has not been implemented. Sprite implements some enhancements to traditional slicing algorithms, most notably in the area of points-to data. Since the tool is not publicly available and the related publications do not deal with this issue, it is not clear how macro dependencies are handled via this approach.

The commercial slicing tool CodeSurfer, marketed by GrammaTech Inc., is probably the most up-to-date slicing program for C/C++ today. It is able to compute various static dependency data by employing the latest code analysis and program slicing technologies. However, it also has modest support for handling preprocessor-related artifacts. It is able to identify the location of macro definitions and uses and present this data to the user. Still, it is not possible to compute slices using macro definitions as criteria. Furthermore, the slices will only include statements that exist after macro expansion. Nevertheless, we used this tool in our experiments because the information supplied by CodeSurfer about macro usage was sufficient to implement our approach.

There are remarkable contributions which offer a solution for development issues: when seeing a macro name in the source code, which macro definitions take part in the expansion. This is the issue of backward macro slices (in our terminology), where the answer requires the analysis of the compilation units in question only. In our approach, however, we use information taken from the whole source code for forward slices as well. (With our approach backward slices can also be computed.)

Livadas and Small identified mappings between the preprocessed and the unpro-

cessed code. The Ghinsu software maintenance environment uses the most similar approach to ours [68]. With this tool by clicking on a macro invocation the called definitions are highlighted (backward macro slice). In addition, it supports both static and dynamic slicing, ripple analysis and other types of program analysis on ANSI compliant C source code. This tool also utilizes a dependency graph where the tokens of preprocessed code are classified according to whether (and if so, how) they are involved in macro expansion. As already mentioned above, it appears that this project has been discontinued, and it had certain drawbacks; for example, certain language features and complex projects consisting of multiple source files are not handled.

The Understand for C++ reverse engineering tool provides cross references between the use and definition of software entities [111]. This includes the step-by-step tracing of macro calls in both directions as well. The user can track back the uses of a given macro definition but the information is imprecise in certain situations. The program fails on the problem depicted in Figure 6.5, and e.g. it misses calls generated by ## operators; or shows a macro call where a parameterized macro name is used without arguments, so no macro expansion occurs.

The above-mentioned APP (Abstract PreProcessor) defines an abstract language. Handling directives in a consistent way allows one to perform an analysis such as slicing as a solution for some preprocessor-related problems [29, 30]. The example presented on slicing is similar to our backward macro slicing, but it has the advantage of indicating the conditional directives in the path. Alas, the implementation drawbacks prevent this tool from being applied to real C programs (e.g. the function-like macros are not supported).

Finally, an interesting topic for future research is the investigation of the so-called dependence clusters [12] on preprocessor slices. A dependence cluster is a set of program statements, all of which are mutually inter-dependent. Dependence clusters are approximated by the set of statements which have similar slice sizes. In the case of combined slicing the presence of similar slice sizes are also observed. Macro slices are, however, usually short, hence the C/C++ slices are dominant in the combined slices. In the preprocessor case, macros with really short macro slices do not necessarily belong to the same cluster. This issue requires more careful investigation, however.

9 Conclusions

Large C and C++ programs exist today that have been in the maintenance phase for many years. The need for tool support is increasing year by year to aid the understanding of large programs with millions of lines of code, and even to aid the implementation of bug-fixes and new features. Our work was dedicated to supporting program maintenance activities impeded by the presence of preprocessor directives. While the preprocessor has increased the productivity in software development, the presence of preprocessor directives has a negative impact in several areas of program maintenance. For instance, there exists one line in the source code of the GCC compiler, which depends on 41 different macros. In addition, after taking into account all the possible configurations this number increases to 187. The risk is high in modifying these kind of preprocessor-dependent points of a program.

Reverse engineering is a process for obtaining facts (relevant information) from legacy programs. Recognizing that after several years of development and operation, the source code is the only relevant and complete documentation of a program, we initially used a source code-based reverse engineering approach. Two crucial aspects of reverse engineering are fact extraction and representation. Here we gave a complete solution for reverse engineering preprocessor-related software artifacts.

Our first contribution was the preprocessor schema (metamodel), which describes our program representation from a preprocessing point of view. The schema represents

the structure of preprocessor directives and also the process of preprocessing with a step-by-step macro expansion. A preprocessor tool was implemented within the Columbus framework, which generates schema instance graphs obtained from the program being analyzed. The tool is called CANPP, and is capable of analyzing industrial size software projects with millions of lines of code.

The preprocessor schema and the API, which provides access to schema instances, allows one to use the detailed information for further analysis purposes in program comprehension, like macro folding or for investigating the include hierarchy. Interoperation with other tools is also facilitated by the XML exports of instance graphs.

Our further contributions were built upon the schema and on processing schema instances. The refactoring of preprocessor directives is barely mentioned in the literature, although refactoring C/C++ programs is a frequent topic. We contributed viewpoints for an elaboration of concrete macro refactorings based on higher level refactoring concepts. We designed a tool architecture, which was implemented mainly based on existing tools, capable of planning, performing and checking refactorings on macros. The usability of the schema was also demonstrated by the developed schema instance exporter, which supported the tool integration with a model transformation system. The proposed method was demonstrated via an elaboration of concrete, applicable refactorings and experiments on real-life programs where macro refactoring was performed at every appropriate program point.

Change impact analysis seeks to provide answers to a central question in maintenance: what parts of a program are affected by a particular change? A well-known method for aiding impact analysis is called program slicing. Slicing was originally introduced to assist debugging, where a set of program points is sought for, which affect the variables of interest at a chosen program point, called the slicing criterion. The area of slicing is fairly diverse, and today there exist a lot of slicing methods and strategies. Their common attribute, however, is not to consider preprocessor macros as program points, the basic unit of slicing. An extensively used approach is when a so-called PDG or SDG (Program or System Dependence Graph) is built in order to compute dependency-based slices. We introduced the novel approach of dependency-based macro slicing in two steps. First, the notion of the Macro Dependence Graph (MDG) was outlined using the macro call relation, and forward and backward macro slices were defined on the MDG. Using macro slices we could tackle questions which could not be answered with traditional C/C++ slicing methods. E.g. which parts of

the source code are affected by a change in a macro body? Second, we integrated dependence graphs and defined connection points to extend traditional C/C++ slices with macro slices. The definitions of combined dependence graph and combined slices were also given. Forward and backward slicing algorithms used to calculate slices were listed as well. We proposed a tool architecture for the global computation of combined slices, and novel slicing notions introduced in our work were validated by experiments. The schema instances served as the MDG, and our macro slicer tool being implemented within the Columbus framework. Combined slices were computed via the integration and extension of existing slicer tools. Both macro slices and combined slices were empirically evaluated based on experiments on real-world programs.

The detailed schema opens up possibilities in several research areas. Generating static instances is a direction where a number of configuration-related issues are waiting to be solved. In our macro refactoring solution, quantitative properties should be improved. The propagation of model-level changes to the source code is still an open issue.

Program comprehension and development could be aided by the intelligent visualization of macro constructs. In this area we have already made progress by extending the Visual Studio plugin with graphical features. We would like to see the notion of macro slicing incorporated into popular slicer tools like CodeSurfer. Experiments in combining dynamic C/C++ slicing with macro slicing would also be helpful.

In conclusion, we introduced novel methods in aiding the maintenance tasks of preprocessed languages. Theoretical and practical results were achieved, and several tools were implemented. Experiments were then performed to prove the practical utility of theoretical results in the areas of modelling, refactoring and slicing preprocessed languages.

Appendices

A

Summary in English

Introduction

Software maintenance requires activities other than the usual ones in the development phase. Maintenance does not consist of just bug fixing in a running program; such an activity may be any modification of a software product after delivery to improve performance or other attributes, or to adapt the product to a changed environment. The need for tool support is increasing year by year to help us better understand large programs with millions of lines of code, and even more to aid the implementation of bug-fixes and new features. Our work was dedicated to supporting program maintenance activities hindered by the presence of preprocessor directives.

While the preprocessor has increased the productivity in software development, the presence of preprocessor directives has a negative impact in several areas of program maintenance. For instance, there exist a line in the source code of the GCC compiler which depends on 41 different macros. The risk is high in modifying such preprocessor-dependent points of a program. Reverse engineering, a fundamental element in supporting maintenance, is a process for obtaining facts (relevant information) from legacy programs. Recognizing that after several years of development and operation, the source code is the only relevant and complete documentation of a program, we used a source code-based reverse engineering approach. We gave a complete solu-

tion for reverse engineering preprocessor-related software artifacts, and which artifacts are then used in program understanding and impact analysis.

Metamodel for the C/C++ preprocessor language

Our first contribution was the preprocessor schema (metamodel), which is a general description of programs from a preprocessing point of view. The schema covers all preprocessor-related elements in a C/C++ source file, and also contains information on preprocessor operations like macro calls. To our knowledge this was the first publicly available general-purpose preprocessor schema. The schema consists of entities with attributes, and their relations, hence it was presented using the UML class diagram notation. A schema instance (model) is a graph that corresponds to a given C/C++ program and contains all the preprocessor-related information in a concrete form. From the schema instance the original source code, the preprocessed source code and all immediate states of the preprocessing process can be obtained. In addition, the schema describes both dynamic (configuration dependent) and static (configuration independent) instances. A preprocessor was implemented within the Columbus framework, which is able to generate schema instance graphs based on the analyzed programs. The tool is called CANPP, and is capable of analyzing industrial size software projects with millions of lines of code. The schema and the related API, which provides access to schema instances, allows one to use the detailed information for further analysis purposes in program comprehension, like implementing macro folding or for investigating the include hierarchy. Therefore the solution is applicable for fully analyzing preprocessor usage at a fine-grained level, and the results were utilized in several research projects. Our further contributions were built upon the schema and on processing schema instances.

Model level refactoring of macros

Refactoring preprocessor directives is barely mentioned in the literature, although refactoring C/C++ programs is a frequent topic. Model level refactoring has the advantage that it formally checks specific conditions, which is necessary when a high level refactoring has many concrete forms. We contributed viewpoints for an elaboration of concrete macro refactorings based on higher level refactoring concepts. Based on

the given criteria, we presented a discussion and elaboration of the refactoring called add parameter for macros. We designed a tool architecture, which was implemented mainly based on existing tools, and is capable of planning, performing and checking refactorings on macros. The usability of the schema was also demonstrated by the developed schema instance exporter, which supported the solid tool integration with a model transformation system. The proposed method is demonstrated via the detailed elaboration of concrete, applicable refactorings and experiments on real-life programs where macro refactoring was performed at every appropriate program point.

Macro slicing

Change impact analysis seeks to find answers to a central question in maintenance: What parts of a program are affected by a particular change? A well-known method for aiding impact analysis is called program slicing. The area of slicing is fairly diverse, and there exist lots of slicing methods and strategies. Their common attribute, however, is not to consider preprocessor macros as program points, the basic unit of slicing. An extensively used approach is when a so-called PDG or SDG (Program or System Dependence Graph) is built in order to compute dependency-based slices. Borrowing ideas from traditional dependency-based slicing, we introduced the novel notion of the Macro Dependence Graph. To ensure appropriate properties for slicing, dependency edges were colored in the graph. Therefore dependence graphs of complete software projects (not just compilation units or individual programs) can be built and used for slicing purposes. We defined both forward and backward type of macro slices, which are computable on the dependence graph.

Combining C/C++ and preprocessor slicing

The use of macro slices is limited to macro constructs, but the real advantage of our approach could be exploited if macro slices could be linked to traditional slices. We combined traditional C/C++ slices with macro slices, giving a more complete dependency set for slicing. The connection points are the places in the source code where (initial) macro calls occur. These points are part of the Macro Dependence Graph, and the resulting tokens from the expanded macro call are part of the dependence graph of C/C++ slicing. We composed a combined dependence graph, on which forward

and backward combined slices were defined. Global forward and backward slicing algorithms were presented as well. These contributions were significant improvements in traditional C/C++ slicing, and were honored with the best paper award in an IEEE conference.

Experimental evaluation of slicing methods

The notions of macro slices and combined slices were evaluated in experiments performed using the relevant tools. We implemented a macro slicer within the Columbus framework which uses schema instances as MDGs. We conducted experiments to evaluate the feasibility of the approach on large-scale programs like the Mozilla, and for getting a better picture of the properties of macro slices. We also proposed a tool architecture for global computation of combined slices. Combined slices were computed via the integration and extension of existing slicer tools. Our macro slicer was used with the CodeSurfer for the C/C++ part, which were integrated using a slice combiner tool. Both macro slices and combined slices were thoroughly evaluated based on experiments on real-world programs. We found that macro slices were significantly smaller than static C++ slices of the same source code (the difference is larger in the case of forward slices). Despite being smaller, macro slices can provide a real improvement since they provide precise information owing to their dynamic nature.

Conclusions

We introduced novel methods to aid the maintenance tasks of preprocessed languages. Both theoretical and practical results were achieved, and several tools were implemented. Experiments were performed to demonstrate the practical usefulness of the theoretical results in the areas of modelling, refactoring and slicing preprocessed languages. There are several promising research areas for future work, the most interesting being the generation of static schema instances for configuration-related analysis.

B

Magyar nyelvű összefoglaló

Bevezető

A szoftver karbantartás másféle tevékenységeket követel meg, mint amelyeket a fejlesztési fázisban már megszoktunk. A karbantartás nem pusztán az átadás után felfedezett hibák javítását jelenti egy már futó programban, karbantartási tevékenység lehet bármely változtatás egy leszállított szoftvertermékben, mely a teljesítményét növeli vagy bármely más tulajdonságát fejleszti, illetve mely a rendszer a megváltozott követelményekhez való adaptációját célozza meg. Évről-évre növekszik az igény olyan eszközök kifejlesztésére, melyek segítenek az ipari méretű, több millió soros programok megértésében. Sőt, méginkább igény mutatkozik olyan eszközökre melyek támogatják a konkrét hibajavításokat és az új funkciók kifejlesztését. Kutatómunkánkat a preprocesszált nyelvi környezetben végzett szoftver karbantartási tevékenységek támogatásának szenteltük.

Amíg a preprocesszor növeli a szoftverfejlesztés hatékonyságát, a direktívák jelenléte negatív hatással van a szoftver karbantartás több területére is. Például a GCC fordító forráskódjában található egy olyan programsor, mely 41 preprocesszor makrótól is függ. Bármilyen változtatás az ilyen mértékű preprocesszoros függésben lévő programsorokban igen magas kockázattal jár. A karbantartás támogatásának alapvető eleme a visszatervezés (reverse engineering). A visszatervezési folyamat célja hogy releváns in-

formációt (tényeket) nyerjen ki már működő programokból. Felismerve, hogy több éves fejlesztés és üzemeltetés után bármely program egyetlen teljes és releváns dokumentációja maga a forrásakód, munkánk során forráskód-alapú visszatervezési megközelítést alkalmaztunk. Teljeskörű megoldást kínálunk preprocesszorral kapcsolatos információk kinyeréséhez (visszatervezéséhez), mely információkat a program megértés, az újratervezés és a hatásanalízis támogatására használunk fel.

Metamodell a C/C++ preprocesszor nyelvhez

Első eredményünk a séma (metamodell) a preprocesszor nyelvhez, mely a programok általános leírása a preprocesszálási folyamat szemszögéből. A séma tartalmaz minden preprocesszorral kapcsolatos elemet ami egy C/C++ forrás állományban található, emellett információt tartalmaz a preprocesszor műveleteiről is, mint például a makró kifejtés. Tudomásunk szerint ez az első nyilvános, általános célú séma a preprocesszorhoz. A séma attribútumokkal ellátott entitásokból, és a köztük lévő kapcsolatokból áll, így az UML osztálydiagram jelölést követve adjuk közre. Séma példányon (modell) egy gráfot értünk, mely egy adott C/C++ programhoz tartozik és tartalmaz minden preprocesszorral kapcsolatos konkrét információt. A séma példányokból mind az eredeti forráskód, mind a preprocesszált forráskód, valamint a preprocesszálási folyamat összes köztes állapota kinyerhető. A séma leírja mind a dinamikus (konfigurációfüggő, egy adott preprocesszor lefutást jellemző), mind a statikus (konfigurációtól független) példányokat is.

A Columbus keretrendszer részeként implementáltunk egy preprocesszort, mely előállítja az elemzett programokhoz tartozó séma példányokat (jelenleg csak dinamikus példányok állíthatók elő az eszköz segítségével). Az eszköz (CANPP) képes több millió nem-üres programsorból álló, ipari méretű szoftverek teljes elemzésére is. A séma és a hozzá tartozó programozói API, mely hozzáférést biztosít a séma példányokhoz, együttes segítségével bárki felhasználhatja a részletes adatokat további elemzések elvégzésére a program megértés területén, mint például a makró folding (makrók szöveges elrejtése és felfedése a fejlesztőkörnyezetben) vagy az include hierarchia további elemzése. A bemutatott megoldás tehát alkalmas a preprocesszor használat részletes és teljes körű elemzésére. Az eredményeket több kutatási projekt során is felhasználtuk, és a további tézisek is nagy mértékben támaszkodnak a sémára és a séma példányok feldolgozására.

Makrók modell szintű újraszervezése

Annak ellenére, hogy a C/C++ programok újraszervezése (refactoring) gyakori téma a szakirodalomban, preprocesszor direktívák újraszervezéséről alig található publikáció. A modell szintű újraszervezés előnye, hogy a modellek különböző feltételek formális ellenőrzésen is átesnek, ami különösen fontos amikor egy magas szintű újraszervezési művelet (refactoring) több konkrét formája is létezhet. Első hozzájárulásunk megfelelő szempontok összegyűjtése makrókkal kapcsolatos magasabb szintű újraszervezési minták konkrét kidolgozásához. A megadott szempontok alapján részletesen is kidolgoztuk és tárgyaltuk a makrókhoz új paraméter hozzáadását célzó újraszervezési műveletet. Eszköz architektúrát terveztünk, melynek implementációja – főleg meglévő eszközökre támaszkodva – alkalmas makrók újraszervezésének tervezésére, végrehajtására és annak ellenőrzésére. A preprocesszor séma használhatóságát bizonyítja az is, hogy a séma példányok konvertálását végző eszköz elkészítésével könnyen megoldhatóvá vált az integráció a modell transzformációs rendszerrel. A bemutatott módszert demonstrálandó konkrét, végrehajtható újraszervezéseket dolgoztunk ki, illetve valós programokon végeztünk kísérleteket, ahol az újraszervezést végrehajtottuk a programok minden arra alkalmas pontján.

Makró szeletelés

A változás-hatásanalízis a szoftver karbantartás egyik központi kérdésére keres választ: a program mely részeire lehet kihatással egy adott változtatás? Egy jól ismert módszer a hatásanalízis támogatására a programszeletelés. A szeletelés területe igen szerteágazó, sok módszer és szeletelési stratégia létezik. Ezek közös tulajdonsága azonban, hogy a preprocesszor makrókat általában nem tekintik program pontoknak, ami a szeletelés alapegysége. A függőség alapú szeletelési módszerek alkalmazásakor egy ún. Program Függőségi Gráf vagy Rendszer Függőségi Gráf épül (továbbiakban PDG vagy SDG – Program/System Dependence Graph) a szeletek kiszámításához. A tradicionális függőség alapú szeletelés ötleteit kölcsönözve megalkottuk a Makró Függőségi Gráfot (továbbiakban MDG – Macro Dependence Graph). A gráfban a függőségi élek színezése biztosítja a szeleteléshez szükséges tulajdonságokat, így teljes szoftver projektek függőségi gráfja is felépíthető szeletelési célokra, nem csak egyes fordítási egységeké. A függőségi gráfon definiáltuk az előrehaladó és hátrahaladó szeleteket, ezzel teljessé tettük a makró szeletelés fogalmát.

C/C++ nyelvű és preprocesszor szeletek összekapcsolása

A makró szeletelés hatásköre önmagában csak a makrókra terjed ki, de a módszer igazi lehetőségeit akkor tudjuk kiaknázni, ha mind a makró szeleteket mind a hagyományos C/C++ szeleteket felhasználjuk. Következő eredményünk hogy a hagyományos C/C++ szeleteket összekapcsoltuk (kombináltuk) a makró szeletekkel, ezzel méginkább teljessé téve a szeletelés alapját képező függőségi halmazt. A kapcsolópontok a forráskódban azok a helyek ahol makró hívás történik. Ezek a pontok részét képezik a Makró Függőségi Gráfnak, emellett a hívás eredményeként keletkező kódrészlet már a C/C++ szeletelés függőségi gráfjának része. Megalkottuk az Összekapcsolt (Kombinált) Függőségi Gráfot, melyen szintén definiáltuk az összekapcsolt előrehaladó és hátrahaladó szeletek fogalmát. Továbbá algoritmusokat is megadtunk előrehaladó és hátrahaladó szeleteket globális számításához. Hozzájárulásunk jelentős előrelépést jelent a hagyományos C/C++ szeletelés területén is, kapcsolódó közleményünk elnyerte a konferencia legjobb cikke díjat.

Szeletelési módszerek gyakorlati kiértékelése

A makró szeletek és összekapcsolt szeletek fogalmát és alkalmazhatóságát a megfelelő eszközökkel végzett kísérletek során értékeltük ki. A Columbus keretrendszer részeként implementáltunk egy makró szeletelő eszközt, mely a preprocesszor séma példányokat használja MDG-ként. Sikeres kísérleteket végeztünk a módszer nagyméretű programokon való alkalmazhatóságát célozva, ahol a Mozilla forráskódján szeleteket számolva valós képet kaptunk a szeletek tulajdonságairól. Emellett eszköz architektúrát terveztünk összekapcsolt szeletek globális számításához. A szeletek számítását meglévő eszközök integrálása és kibővítése útján oldottuk meg. A már említett makró szeletelő eszközünket, és a CodeSurfer C/C++ szeletelő eszközt felhasználva, ezeket egy szelet összekapcsoló eszközzel integrálva valósítottuk meg a szeletelést. A makró szeletek és az összekapcsolt szeletek is kiértékelésre kerültek valós programokon végrehajtott kísérletek során. Azt tapasztaltuk, hogy a makró szeletek lényegesen kisebbek mint a statikus C++ szeletek ugyanazon a programon mérve, különösen előremutató szeletek esetében. Bár a makró szeletek kisebbek, mégis jelentős fejlődést jelentenek mivel dinamikus természetüknél fogva pontos információval bővítik a C/C++ szeleteket.

Konklúzió

A dolgozatban preprocesszált nyelvi környezetben fejlesztett szoftverek karbantartását támogató újszerű módszereket mutattunk be. Céljainkat mind alapvető, elméleti eredményeket felmutatva, mind ezek alkalmazására készült eszközök implementálásával értük el. Az elméleti eredmények használhatóságának alátámasztására kísérleteket végeztünk preprocesszált nyelvek forráskódjának modellezése, újraszervezése és szeletelése területén. A munka folytatásához több ígéretes kutatási irány is adott, ezek közül talán a legérdekesebb a statikus séma példányok előállítása konfigurációkkal kapcsolatos elemzések végzéséhez.

C

Further details on the preprocessor schema

C.1 Sample PPML output

PPML (PreProcessor Markup Language) is the XML representation of the schema instances. The structure of the schema is followed by the XML elements and attributes. A sample piece of source code can be seen in Listing C.1. The corresponding PPML code is shown in Listing C.2. Note that in the case of PPML, the line and column information of nodes (which is important for each node) has been omitted from the listing due to space constraints.

```
1 #if VERBOSE >= 2
2   print("trace␣message");
3   #endif
4 #define PRETTY_PRINT(msg) printf(msg);
5 if (n < 10)
6   PRETTY_PRINT("n␣is␣less␣than␣10");
7 else
8   PRETTY_PRINT("n␣is␣at␣least␣10");
```

Listing C.1: PPML sample: source code fragment

```
<Project id='id100' ork='1'>
<File id='id101' name='example.cpp'>
  <If id='id318' enabled='false'>
    <DirectiveText id='id319' name='␣'/>
    <DirectiveId id='id320' name='VERBOSE'>
    </DirectiveId>
    <DirectiveText id='id321' name='␣'/>
    <DirectiveText id='id322' name='&gt;='/>
    <DirectiveText id='id323' name='␣'/>
    <DirectiveText id='id324' name='2'/>
  </If>
  <Endif id='id325'>
    <dependsOn ref='id318'/>
  </Endif>
  <FuncDefine id='id326' name='PRETTY_PRINT' isExternal='false'>
    <defineRef ref='id336'/>
    <defineRef ref='id343'/>
    <Parameter id='id328' name='msg'/>
    <DirectiveId id='id329' name='printf'>
    </DirectiveId>
    <DirectiveText id='id330' name='('/>
    <DirectiveId id='id331' name='msg'>
      <refersToParameter ref='id328'/>
    </DirectiveId>
    <DirectiveText id='id332' name=')'/>
    <DirectiveText id='id333' name=';'/>
  </FuncDefine>
  <Text id='id334' name='if␣(n␣&lt;␣10)&#10;␣␣'>
  </Text>
  <Id id='id335' name='PRETTY_PRINT'>
    <defineRef ref='id336'/>
  </Id>
  <Text id='id337' name='('>
  </Text>
  <Text id='id339' name='&quot;n␣is␣less␣than␣10&quot;'>
  </Text>
  <Text id='id340' name=')'>
  </Text>
  <Text id='id341' name=';&#10;else&#10;␣␣'>
  </Text>
  <Id id='id342' name='PRETTY_PRINT'>
    <defineRef ref='id343'/>
  </Id>
  <Text id='id344' name='('>
  </Text>
  <Text id='id346' name='&quot;n␣is␣at␣least␣10&quot;'>
  </Text>
  <Text id='id347' name=')'>
  </Text>
  <Text id='id348' name=';&#10;'>
  </Text>
</File>
</Project>
<FuncDefineRef id='id336' ork='1'>
  <refersToId ref='id335'/>
  <refersToDefinition ref='id326'/>
  <Argument id='id338' line='6' col='16' endLine='6' endCol='34'>
    <consistsOf ref='id339'/>
  </Argument>
</FuncDefineRef>
<FuncDefineRef id='id343' ork='1'>
```

```
    <refersToId ref='id342'/>
    <refersToDefinition ref='id326'/>
    <Argument id='id345'  line='8' col='16' endLine='8' endCol='33'>
      <consistsOf ref='id346'/>
    </Argument>
  </FuncDefineRef>
```

Listing C.2: PPML sample: the corresponding XML code

C.2 CANPP command line options

Table C.1 below summarizes the most important command line options of our preprocessor implementation.

Option	Description
-P	Preprocess to file (default)
-E	Preprocess to stdout
-EP	Preprocess to stdout, no #line
-s	File name for schema instance
-version	Print version information
-dPPML	Generate PPML (XML) output
-dGXL	Generate GXL (XML) output
-I	Add to include search path
-D	Define macro, format: -D<name>[<params>][=|#<value>]
-U	Remove predefined macro
-u	Remove all predefined macros
-X	Ignore standard places
-nostdinc	Ignore standard places
-FI	Forced include file (Microsoft)
-include	Forced include file (GCC)
-C	Do not strip comments
-nC	No comments
-npsi	Do not write .psi output
-w	Disable all warnings
-dM	Dump macros
-dep	Create dependency list
-di	Generate i file from schema
-mc	Create macro calls structure
-GR	MS - Enable Run-Time Type Information (defines macro _CPPRTTI)
-MTd	MS - Use Run-Time Library (defines macro _MT, _DEBUG)
-MDd	MS - Use Run-Time Library (defines macro _MT, _DLL, _DEBUG)
-ini	Use the given ini file for C++ files

Table C.1: CANPP command line options

Bibliography

[1] Bram Adams, Wolfgang De Meuter, Herman Tromp, and Ahmed E. Hassan. Can we refactor conditional compilation into aspects? In *AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development*, pages 243–254, New York, NY, USA, 2009. ACM.

[2] Bram Adams, Bart Van Rompaey, Celina Gibbs, and Yvonne Coady. Aspect mining in the presence of the c preprocessor. In *LATE '08: Proceedings of the 2008 AOSD workshop on Linking aspect technology and evolution*, pages 1–6, New York, NY, USA, 2008. ACM.

[3] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with dynamic slicing and backtracking. *Software – Practice and Experience (SPE)*, 23(6):589–616, 1993.

[4] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM PLDI 1990*, pages 246–256, New York, NY, USA, 1990. ACM Press.

[5] Lerina Aversano, Massimiliano Di Penta, and Ira D. Baxter. Handling preprocessor-conditioned declarations. In *Proceedings of SCAM 2002*, pages 83–92, 2002.

[6] Greg J. Badros and David Notkin. A framework for preprocessor-aware C source code analyses. *Software - Practice and Experience*, 30(8):907–924, 2000.

[7] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *WCRE '01: Proceedings of the Eighth Working Conference on Reverse Engineering (WCRE'01)*, page 281, Washington, DC, USA, 2001. IEEE Computer Society.

[8] Bell Canada Inc., Montréal, Canada. *DATRIX – Abstract semantic graph reference manual*, version 1.2 edition, January 2000.

[9] Árpád Beszédes, Csaba Faragó, Zsolt Mihály Szabó, János Csirik, and Tibor Gyimóthy. Union slices for program maintenance. In *Proceedings of ICSM 2002*, pages 12–21. IEEE Computer Society, October 2002.

[10] Árpád Beszédes, Tamás Gergely, and Tibor Gyimóthy. Graph-less dynamic dependence-based dynamic slicing algorithms. In *Proceedings of SCAM 2006)*, pages 21–30, September 2006.

[11] David Binkley and Mark Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of ICSM 2003*, pages 44–53. IEEE Computer Society, September 2003.

[12] David Binkley and Mark Harman. Locating dependence clusters and dependence pollution. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM'05)*, pages 177–186. IEEE Computer Society, September 2005.

[13] David W. Binkley. The application of program slicing to regression testing. *Information and Software Technology*, 40(11-12):583–594, 1998.

[14] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, 1996.

[15] Paolo Bottoni, Francesco Parisi-Presicce, and Gabriele Taentzer. Specifying integrated refactoring with distributed graph transformations. *Lecture Notes in Computer Science*, 3062:220–235, 2004.

[16] Fabian Büttner and Martin Gogolla. Realizing graph transformations by pre- and postconditions and command sequences. In *ICGT*, pages 398–413, 2006.

[17] C Compatible Preprocessor for PHP. `http://code.metala.org/p/ccpp/`, 2009.

[18] E. J. Chikofsky and J. H. Cross II. Reverse Engineering and Design Recovery: A Taxonomy. In *IEEE Software 7*, pages 13–17, January 1990.

[19] Aniello Cimitile, Andrea de Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *Journal of Software Maintenance: Research and Practice*, 8(3):145–178, 1996.

[20] Anthony Cox and Charles L. A. Clarke. Relocating xml elements from preprocessed to unprocessed code. In *IWPC*, pages 229–238, 2002.

[21] G. Csertan, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and Dániel Varró. Viatra - visual automated transformations for formal verification and validation of uml models. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 267–270, 2002.

[22] Jörg Czeranski, Thomas Eisenbarth, Holger M. Kienle, Rainer Koschke, Erhard Plödereder, Daniel Simon, Yan Zhang V, Jean-Francois Girard, and Martin Würthner. Data exchange in Bauhaus. In *WCRE*, pages 293–295, 2000.

[23] Thomas R. Dean, Andrew J. Malton, and Ric Holt. Union Schemas as a Basis for a C++ Extractor. In *Proceedings of WCRE'01*, pages 59–67, October 2001.

[24] Jürgen Ebert, Bernt Kullbach, Volker Riediger, and Andreas Winter. GUPRO - Generic Understanding of Programs. In Tom Mens, Andy Schürr, and Gabriele Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2002.

[25] Michael D. Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. *IEEE Transactions on Software Engineering*, 28(12), Dec 2002.

[26] ExtUtils::PerlPP - A Perl Preprocessor. `http://search.cpan.org/~jwied/Msql-Mysql-modules-1.2219/lib/ExtUtils/PerlPP.pm`, 2001-2009.

[27] Richard Fanta and Vaclav Rajlich. Reengineering object-oriented code. In *ICSM '98: Proceedings of the International Conference on Software Maintenance*, page 238, Washington, DC, USA, 1998. IEEE Computer Society.

[28] Jean-Marie Favre. The cpp paradox. In *9th European Workshop on Software Maintenance*, 1995.

[29] Jean-Marie Favre. Preprocessors from an abstract point of view. In *Proceedings of the International Conference on Software Maintenance (ICSM 1996)*, page 329, 1996.

[30] Jean-Marie Favre. Cpp denotational semantics. In *Proceedings of SCAM 2003*, page 22, 2003.

[31] Rudolf Ferenc and Árpád Beszédes. Data Exchange with the Columbus Schema for C++. In *Proceedings of the 6th European Conference on Software Maintenance and Reengineering (CSMR 2002)*, pages 59–66. IEEE Computer Society, March 2002.

[32] Rudolf Ferenc, Árpád Beszédes, and Tibor Gyimóthy. Fact Extraction and Code Auditing with Columbus and SourceAudit. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, page 513. IEEE Computer Society, September 2004.

[33] Rudolf Ferenc, Árpád Beszédes, Mikko Tarkiainen, and Tibor Gyimóthy. Columbus - reverse engineering tool and schema for c++. In *Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, October 2002.

[34] Rudolf Ferenc, Ferenc Magyar, Árpád Beszédes, Ákos Kiss, and Mikko Tarkiainen. Columbus – tool for reverse engineering large object oriented software systems. In *Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001)*, pages 16–27. University of Szeged, June 2001.

[35] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting Facts from Open Source Software. In *Proceedings of the 20th International Conference on Software Maintenance (ICSM 2004)*, pages 60–69. IEEE Computer Society, September 2004.

[36] Rudolf Ferenc, Susan Elliott Sim, Richard C Holt, Rainer Koschke, and Tibor Gyimóthy. Towards a Standard Schema for C/C++. In *WCRE 2001*, pages 49–58. IEEE Computer Society, October 2001.

[37] M. Fowler. *Refactoring Improving the Design of Existing Code*. Addison-Wesley, 2002.

[38] Homepage of FrontEndART Ltd. `http://www.frontendart.com`.

[39] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[40] Alejandra Garrido. Program refactoring in the presence of preprocessor directives. Ph.D. thesis, University of Illinois at Urbana-Champaign, USA, october 2005.

[41] Alejandra Garrido and Ralph Johnson. Challenges of refactoring C programs. In *Proceedings of IWPSE 2002*, pages 6–14. ACM, 2002.

[42] Alejandra Garrido and Ralph Johnson. Refactoring c with conditional compilation. In *ASE*, pages 323–326, 2003.

[43] Alejandra Garrido and Ralph Johnson. Analyzing multiple configurations of a c program. In *Proceedings of ICSM 2005*, pages 379–388. IEEE Computer Society, 2005.

[44] GNU Compiler Collection Homepage. `http://gcc.gnu.org`.

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

[46] Martin Gogolla, Fabian Büttner, and Duc-Hanh Dang. From graph transformation to OCL using USE. In *AGTIVE*, pages 585–586, 2007.

[47] Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-based specification environment for validating UML and OCL. *Sci. Comp. Program.*, 69(1-3):27–34, 2007.

[48] Katsuhiko Gondow, Hayato Kawashima, and Takashi Imaizumi. Tbcppa: A tracer approach for automatic accurate analysis of c preprocessor's behaviors. In *SCAM*, pages 35–44, 2008. CSMR SPLST.

[49] Homepage of GrammaTech's CodeSurfer. `http://www.grammatech.com/products/codesurfer`, 2009.

[50] Judith Grass and Yih farn Chen. The c++ information abstractor. In *In The Second USENIX C++ Conference*, pages 265–277, 1992.

[51] The GXL Homepage. `http://www.gupro.de/GXL/`.

[52] Ric Holt, Ahmed E. Hassan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/R Schema for the Datrix C/C++/Java Exchange Format. In *Proceedings of WCRE'00*, November 2000.

[53] Ric Holt, Andreas Winter, and Andy Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, November 2000.

[54] Homepage of DMS. `http://www.semdesigns.com/Products/DMS`, 2009.

[55] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[56] International Standards Organization. *Programming languages — C++*, ISO/IEC 14882:1998(E) edition, 1998.

[57] International Standards Organization. *Programming languages — C*, ISO/IEC 9899:1999 edition, 1999.

[58] JavaPP - Java Preprocessor. `http://www.slashdev.ca/javapp/`, 2008.

[59] Does Java Need a Preprocessor? `http://www.javalobby.org/java/forums/t18116.html`, 2005.

[60] Homepage of jFactor. `http://old.instantiations.com/jfactor/`, 2009.

[61] JPP: The Java Pre-Processor. `http://www.geocities.com/h2428/ceco/jpp.html`, 2005.

[62] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of ICSE 1994, 16th International Conference on Software Engineering*, pages 49–57. IEEE Computer Society, 1994.

[63] Bernt Kullbach and Volker Riediger. Folding: An Approach to Enable Program Understanding of Preprocessed Languages. In *Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001)*, pages 3–12. IEEE Computer Society, 2001.

[64] Mario Latendresse. Fast symbolic evaluation of c/c++ preprocessing using conditional values. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 170–179. IEEE Computer Society, March 2003.

[65] Mario Latendresse. Rewrite systems for symbolic evaluation of c-like preprocessing. In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004)*, pages 165–173. IEEE Computer Society, March 2004.

[66] Timothy C. Lethbridge. The dagstuhl middle model: An overview. *Electronic Notes in Theoritical Computer Science*, 94:7–18, 2003.

[67] Tihamér Levendovszky, László Lengyel, Gergely Mezei, and Hassan Charaf. A Systematic Approach to Metamodeling Environments and Model Transformation Systems in VMTS. In *Electronic Notes in Theoretical Computer Science*, pages 65–75, 2005.

[68] P.E. Livadas and D.T. Small. Understanding code containing preprocessor constructs. In *Proceedings of IWPC 1994, Third IEEE Workshop on Program Comprehension*, pages 89–97, Nov 1994.

[69] Bart Van Rompaey Matthias Rieger and Roel Wuyts. Teaching famix about the preprocessor. In *FAMOOSr 2007: 1st Workshop on FAMIX and Moose in Reengineering*, pages 13–16, 2007.

[70] C. A. Mennie and C. L. A. Clarke. Giving meaning to macros. In *Proceedings of IWPC 2004*, pages 79–88. IEEE Computer Society, 2004.

[71] Tom Mens. On the use of graph transformations for model refactoring. In *GTTSE*, pages 219–257, 2006.

[72] Tom Mens, Serge Demeyer, Bart Du Bois, Hans Stenten, and Pieter Van Gorp. Refactoring: Current research and future trends. *Electr. Notes Theor. Comput. Sci.*, 82(3), 2003.

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

[74] Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17:247–276, 2005.

[75] Markus Mock, Darren C. Atkinson, Craig Chambers, and Susan J. Eggers. Program slicing with dynamic points-to sets. *IEEE Transactions on Software Engineering*, 31(8):657–678, 2005.

[76] Microsoft Developer Network Online Library. `http://msdn.microsoft.com`.

[77] Hausi A Müller, Kenny Wong, and Scott R Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proceedings of ACFAS*, 1994.

[78] Object Management Group Inc. *OMG Unified Modeling Language Specification*, version 1.3 edition, 1999.

[79] William F. Opdyke. Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, USA, 1992.

[80] William F. Opdyke and Ralph E. Johnson. Creating abstract superclasses by refactoring. In *CSC '93: Proceedings of the 1993 ACM conference on Computer science*, pages 66–73, New York, NY, USA, 1993. ACM.

[81] Karl J. Ottenstein and Linda M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT-/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE)*, number 19(5) in SIGPLAN Notices, pages 177–184, Pittsburgh, Pennsylvania, May 1984.

[82] Yoann Padioleau. Parsing c/c++ code without pre-processing. In *CC '09: Proceedings of the 18th International Conference on Compiler Construction*, pages 109–125, Berlin, Heidelberg, 2009. Springer-Verlag.

[83] PICA - Perl Installation and Configuration Agent. `http://pica.sourceforge.net/`, 2004-2009.

[84] pyp - Simple Preprocessor for Python. `http://www.freenet.org.nz/python/pyp/`, 2009.

[85] pypp - Python Preprocessor. `http://code.google.com/p/pypp/`, 2009.

[86] Václav Rajlich. A model for change propagation based on graph rewriting. In *Proceedings of ICSM 1997)*, pages 84–91, October 1997.

[87] Václav Rajlich and Prashant Gosavi. Incremental change in object-oriented programming. *IEEE Software*, 21(4):62–69, 2004.

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

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

[90] László Vidács (supervisor) Richárd Dévai. Visualization of preprocessor dependencies. Master's thesis, University of Szeged, Hungary (in Hungarian), 2009.

[91] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for Smalltalk. *Theor. Pract. Object Syst.*, 3(4):253–263, 1997.

[92] Gregg Rothermel and Mary Jean Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of ISSTA'94*, pages 169–183, August 1994.

[93] Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-clr: a tool for navigating highly configurable system software. In *ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 9, New York, NY, USA, 2007. ACM.

[94] Nieraj Singh, Graeme Johnson, and Yvonne Coady. Cvime: viewing conditionally compiled c/c++ sources through java. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 730–731, New York, NY, USA, 2006. ACM.

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

[96] Gregor Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Trans. Softw. Eng. Methodol.*, 5(2):146–189, 1996.

[97] Gregor Snelting. Software reengineering based on concept lattices. In *CSMR*, pages 3–10, 2000.

[98] Stéphane S. Somé and Timothy Lethbridge. Parsing minimization when extracting information from code in the presence of conditional compilation. In *IWPC*, pages 118–125, 1998.

[99] H. Spencer and G. Collyer. #ifdef considered harmful, or portability experience with C News. In *USENIX Summer Technical Conference*, pages 185–197, June 1992.

[100] Diomidis Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Transactions on Software Engineering*, 29(11):1019–1030, November 2003.

[101] Diomidis Spinellis. A refactoring browser for c. In *ECOOP'08 International Workshop on Advanced Software Development Tools and Techniques (WASDeTT)*, 2008.

[102] Diomidis Spinellis. Optimizing header file include directives. *Journal of Software Maintenance and Evolution: Research and Practice*, 22, 2010.

[103] sqlpp - SQL preprocessor. `http://search.cpan.org/~karasik/bin-sqlpp-0.06/bin/sqlpp`, 2009.

[104] R. M. Stallman and Z. Weinberg. *The C Preprocessor, A GNU Manual*, 2001.

[105] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Third edition, 1997.

[106] Sunifdef homepage. `http://www.sunifdef.strudl.org/`, 2008.

[107] Andrew Sutton and Jonathan I. Maletic. How we manage portability and configuration with the c preprocessor. In *ICSM*, pages 275–284, 2007.

[108] Gabriele Taentzer, Dirk Müller, and Tom Mens. Specifying domain-specific refactorings for AndroMDA based on graph transformation. In *AGTIVE*, pages 104–119, 2007.

[109] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, September 1995.

[110] Lance Tokuda and Don Batory. Evolving object-oriented designs with refactorings. *Automated Software Enginering*, 8(1):89–120, 2001.

[111] Understand for C++ homepage. `http://www.scitools.com`, 2007.

[112] Homepage of the Unravel project. `http://www.itl.nist.gov/div897/sqg/unravel/unravel.html`, 2008.

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

[114] Marian Vittek. Refactoring browser with preprocessor. In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003)*, pages 101–110, Benevento, Italy, March 2003.

[115] Marian Vittek, Peter Borovanský, and Pierre-Etienne Moreau. A collection of c, c++ and java code understanding and refactoring plugins. In *ICSM 2005 Industrial and Tool Volume*, pages 61–64, 2005.

[116] Marian Vittek, Peter Borovansky, and Pierre-Etienne Moreau. A C++ refactoring browser and method extraction. In *Software Engineering Techniques: Design for Quality: IFIP TC-2 Working Conference on Software Engineering Techniques - SET 2006*, volume 227, pages 325–336. Springer, 2006.

[117] Kiem-Phong Vo and Yih-Farn Chen. Incl: A tool to analyze include files. In *USENIX Summer Technical Conference*, pages 199–208, June 1992.

[118] K. Wansbrough. Macros and preprocessing in Haskell. `http://www.cl.cam.ac.uk/~kw217/research/misc/hspp-hw99.ps.gz`, 1999.

[119] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

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

[121] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *ACM SIGSOFT Softw. Eng. Notes*, 30(2):1–36, 2005.

[122] Jianjun Zhao. A slicing-based approach to extracting reusable software architectures. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering (CSMR'00)*, pages 215–223, February 2000.

References

[VB03] László Vidács and Árpád Beszédes. Opening up the C/C++ preprocessor black box. In *Proceedings of SPLST 2003, 8th Symposium on Programming Languages and Software Tools*, pages 45–57, June 2003.

[VBF04] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Columbus Schema for C/C++ Preprocessing. In *Proceedings of CSMR 2004 (8th European Conference on Software Maintenance and Reengineering)*, pages 75–84. IEEE Computer Society, March 2004.

[VBF07] László Vidács, Árpád Beszédes, and Rudolf Ferenc. Macro impact analysis using macro slicing. In *Proceedings of ICSOFT 2007, Second International Conference on Software and Data Technologies*, pages 230–235, July 2007.

[VBG09] László Vidács, Árpád Beszédes, and Tibor Gyimóthy. Combining preprocessor slicing with C/C++ language slicing. *Science of Computer Programming*, 74(7):399–413, May 2009.

[VGF06] László Vidács, Martin Gogolla, and Rudolf Ferenc. From C++ Refactorings to Graph Transformations. *Electronic Communications of the EASST (ICGT 2006 Workshop Software Evolution through Transformations)*, 3:127–141, September 2006.

[Vid09] László Vidács. Refactoring of C/C++ Preprocessor constructs at the model level. In *Proceedings of ICSOFT 2009, 4th International Conference on Software and Data Technologies*, pages 232–237, July 2009.

[VJBG08] László Vidács, Judit Jász, Árpád Beszédes, and Tibor Gyimóthy. Combining preprocessor slicing with C/C++ language slicing. In *Proceedings of ICPC 2008, 16th IEEE International Conference on Program Comprehension*, pages 163–171. IEEE Computer Society, June 2008. Best paper award.