Software Maintenance Methods for Preprocessed Languages

Summary of the PhD Dissertation

by

László Vidács

Supervisor:
Dr. Tibor Gyimóthy

PhD School in Computer Science
Institute of Informatics
University of Szeged

Szeged 2009

Introduction

A large amount of effort has been made in supporting the software development process. The best-known methods and methodologies support the development phase of the software life-cycle. The increased productivity in development and the constantly changing technologies introduced the notion of software maintenance. Large C and C++ programs exist today that have been in the maintenance phase for many years. Owing to its nature, software maintenance requires activities other than the usual ones in the development phase. Maintenance does not consist of just bug fixing in an operating 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. Furthermore, maintenance costs can easily be underestimated. Correcting defects, keeping the software up to date with the changing 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.

Program understanding is a crucial part of any maintenance tasks. The developers who want to modify the existing system need both a high level, overall view of the system and detailed implementation level information. The information extraction process is called reverse engineering [5]. 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 become incomplete or inaccurate over years of software operation. In addition, outdated documentation may mislead the maintainer and create additional costs.

Although development and maintenance are closely 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 a 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 [6]. The view taken by most is just the opposite when software maintenance or program understanding tasks have to be done: the presence of preprocessor directives is always mentioned as an obstacle [17]. Some types of macros and conditionals may be transformed into C/C++ program code [13], and methods have been introduced for removing unnecessary conditional directives [1], but the vast majority of directive uses remain present in the code. 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 work was dedicated to supporting program maintenance activities impeded by the presence of preprocessor directives. Alas, preprocessor issues are often completely neglected by C/C++ analyzer tools, or at least, handled rather poorly (there exist some notable exceptions [7, 12]). 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. Not only is the structure of directives modelled by the schema, but so is the preprocessing operation. A schema instance is a concrete (graph) representation of a program. It

is the result of the reverse engineering process, and conforms with the schema. The information obtained may be used for program understanding purposes like macro folding or extracting include hierarchy.

Moreover, our other contributions were also built upon these results. We contributed viewpoints for an elaboration of concrete macro refactorings based on higher level refactoring concepts. A tool architecture was also designed and implemented for planning, performing and checking refactorings on macros.

Change impact analysis is the study of the ripple effect that is caused by a change in a large software package [3, 16]. We introduced novel methods in the area of program slicing, which is a proper method for change impact analysis purposes. We integrated macro-related analysis for program slicing in two steps. First, the Macro Dependence Graph (MDG) was constructed and forward and backward macro slices were defined. Computing macro slices involves program points which would have been missed without the MDG using traditional C/C++ slicing. However, the real advantage of the MDG can be exploited in the second step, where C/C++ language slices and macro slices are combined. The two types of slices may be combined in both forward and backward directions. We presented definitions of the combined dependence graph and the combined slices, together with algorithms for computing these slices. The proposed slicing methods were implemented and evaluated via experiments on real-life programs.

Our results have been grouped into five contributions, divided into two parts according to the research topics. In the remaining part of the thesis summary, the following contribution points will be presented:

- I/1 Metamodel for the C/C++ preprocessor language
- I/2 Model level refactoring of macros
- II/1 Macro slicing
- II/2 Combining C/C++ language and preprocessor slicing
- II/3 Experimental evaluation of slicing methods

Part I - Modelling and refactoring preprocessor directives

The contributions of the first part are related to preprocessor models extracted from the source code. First we introduce the preprocessor metamodel, which serves as a basis for all maintenance-related methods. Next, our most recent work is presented which investigates the model level refactoring of preprocessor constructs (especially macros) in terms of graph transformations.

I/1 Metamodel for the C/C++ preprocessor language

Columbus Schema for C/C++ preprocessing

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 corresponds to a concrete 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. Therefore the solution is applicable for fully analyzing preprocessor usage at a fine-grained level.

The metamodel is not presented here due to space constraints, but an example piece of source code can be seen in Listing 1, and the corresponding schema instance can be found below in Figure 1. Each preprocessor language element is represented by a node with attributes in the instance graph, e.g. the definition of `__MATHDECL_1` (10), condition for `__STDC__` (20), definition of `__MATH_PRECNAME` (22), the include directive (25) and the included file (27) with its subgraph. Furthermore, steps of the macro substitution can be understood by following the edges of the graph (see reference object (40)).

```
#define __MATHDECL_1(type, function,suffix, args) \ /*ID=10*/
  extern type __MATH_PRECNAME(function,suffix) args __THROW
...
#if defined __USE_MISC || defined __USE_ISOC99
...
#ifdef __STDC__                                     /*ID=20*/
# define __MATH_PRECNAME(name,r) name##f##r         /*ID=22*/
#else
# define __MATH_PRECNAME(name,r) name/**/f/**/r
#endif
#include <bits/mathcalls.h>                         /*ID=25*/
#undef __MATH_PRECNAME
```

Listing 1: Example code from `math.h`

A programming API was 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 [9] and PPML (our XML representation) as well.

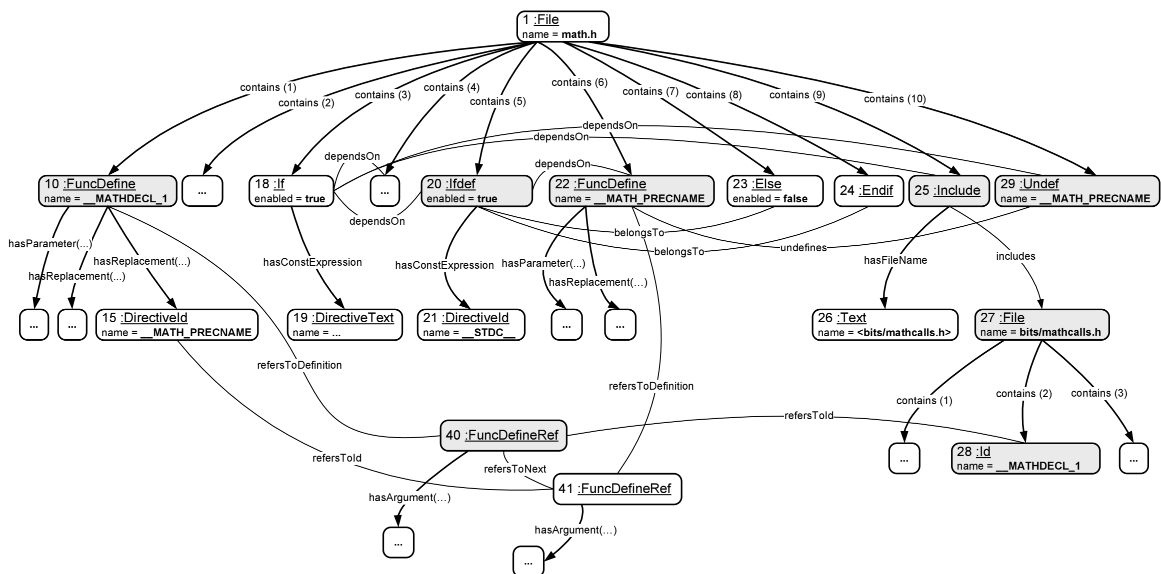

Figure 1: Dynamic schema instance

Building schema instances

We implemented a preprocessor for building schema instances, which is called CANPP. It is part of the Columbus Framework and it can analyze industrial sized software projects with millions of lines of code. It is only capable of building dynamic instances, but this still offers a wide range of possibilities. The preprocessor was 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. The process of reverse engineering directives is depicted in Figure 2 below.

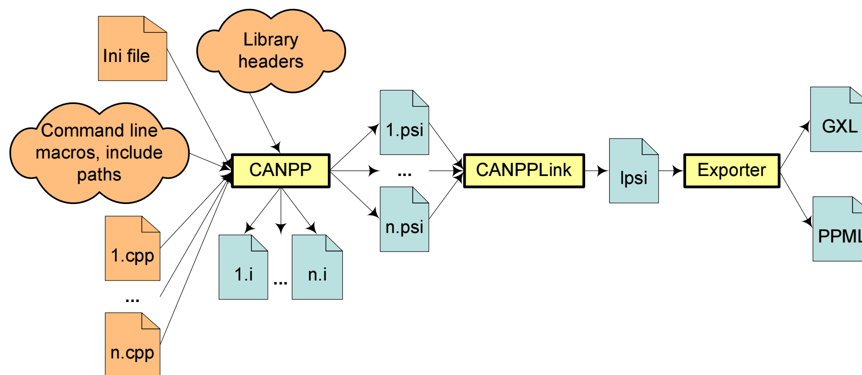

Figure 2: Preprocessor reverse engineering process

Utilization of results

First, all four of the following thesis contributions rely heavily on the schema. Second, there are several applications of the schema instances in the Columbus framework. Preprocessor-related analysis was used to aid the static rule checking capabilities of the SourceAudit tool. Include dependency was used to help in incremental parsing in the Columbus tool. A Visual Studio plugin was developed to implement the macro folding mechanism [11] to show intermediate states of macro replacements for the developer.

Our results were utilized in several successful industrial and academic research projects as well. For instance, a 5MLoc large software project was analyzed using our tool in a joint project with the Nokia Research Center for compile time optimization. Another example here is the OpenOffice++ R&D project co-funded by the EU, where the aim was to analyze and improve the architecture of OpenOffice.org and the quality of its source code.

Own contribution

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 were published in research papers [19, 20].

I/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 related to the preprocessor metamodel. The design of applicable refactorings on macros includes the following considerations. As an alternative, one can always consider defining C/C++ constants or functions, or the possibility of variadic macros. In preprocessing, the environment plays an important role, including the command line defined and standard macros, whose aspect usually changes the preconditions. Similar to other languages, call sites of macros must be traversed and the necessary modifications must be made in order to keep the model consistent. The concrete form of a macro refactoring is also influenced by the type of the macros being transformed (object-like, function-like or variadic types). Based on the determined criteria, we presented a discussion and elaboration of the refactoring called add parameter for macros. We applied a graph transformation approach for refactoring [15, 14] using left-hand side and right-hand side graphs. An example refactoring for adding a parameter to an object-like macro is shown in Figure 3.

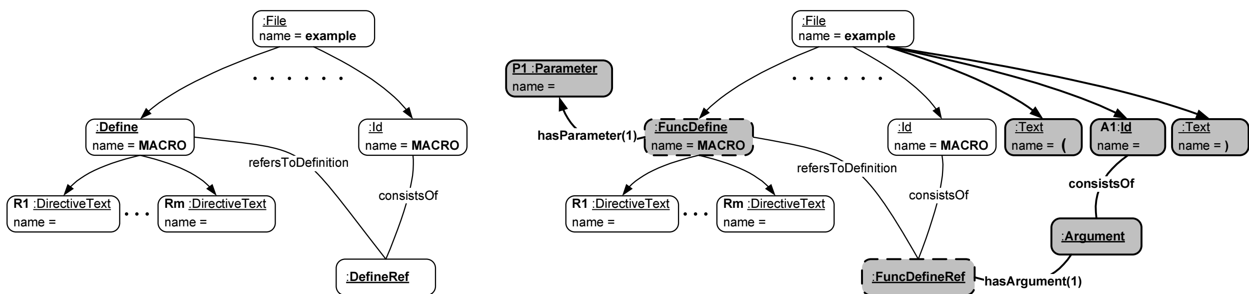

Figure 3: Add parameter to object like macro - left hand side and right hand side of the rule

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. The proposed architecture is shown in Figure 4. The preprocessor metamodel (A) plays an important role in each phase. The transformation rule is designed by adjusting the left and right hand side models in the USE system (B). Based on these models, a rule description file is created by hand (a straightforward step). The Rule2OCL tool uses the metamodel and a rule description to generate applicable rules. OCL pre and postconditions are also automatically generated. The initial program model was produced by our Columbus tool (C). We implemented an exporter which can transform schema instances to an understandable form for the USE UML specification environment, where the transformations are handled [4, 8]. The USE system checks 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. We conducted experiments to justify our proposed method. The object-like macro refactoring was implemented in two steps. The macro definition was changed and a parameter was added first, then each call site was extended with an argument. During the experiments we found that the proposed tool-set was appropriate for medium-sized programs, and also for validating preprocessor models and the metamodel itself.

Own contribution

The above-mentioned contributions are the work of the author, the main results being published in research paper [18]. Some notions used in this contribution point were published in an earlier work in a C++ context [23]. In the earlier work, the C++ metamodel used was not the work of the author, basic notions of model level transformations using the metamodel were shared results, while the elaboration and implementation of C++ refactorings were results of the author.

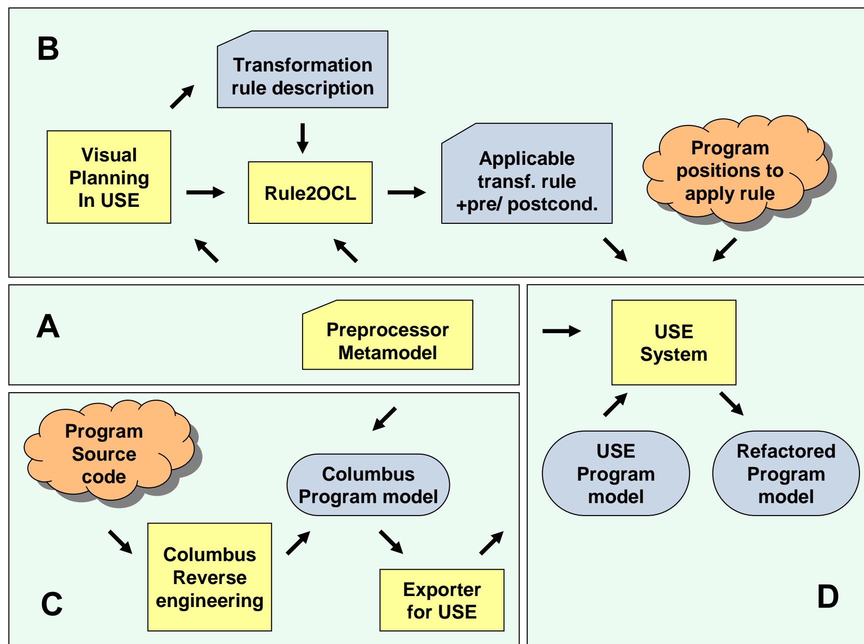

Figure 4: Refactoring architecture

Part II - Slicing methods for change impact analysis

In the second part we our incorporate macro-related analysis into program slicing. The handling of changes is a key issue in software maintenance, change impact analysis being the study of the ripple effect that is caused by a change in a large system. A well-known method for aiding impact analysis is called program slicing [25, 26]. It is an analysis method for extracting parts of a program which represent a specific sub-computation of interest. 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 [10]. We introduced the approach of dependency-based macro slicing in two steps. First, the Macro Dependence Graph was outlined based on the macro call relation, and forward and backward macro slices were also defined. Using macro slices we could tackle questions which could not be answered with traditional C/C++ slicing methods, like "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 combined slices are listed as well. We proposed a tool architecture for the global computation of combined slices. Novel slicing notions, introduced in our work, were validated by experiments. Both macro slices and combined slices were empirically evaluated based on experiments on real-world programs.

As a motivating problem, let us 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, and finally after several replacements become part of C/C++ language constructs. These constructs may affect other parts of the program, which may be captured by traditional C/C++ language slices. 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 slices starting from replaced parts gives us all the affected points.

```
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 2: Motivating example for combined slices

Our idea is illustrated with the help the following piece of source code shown above in Listing 2. The slicing criterion for macro slicing is the macro definition in line 1, and the corresponding macro slice contains lines 1, 3 and 4. 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 for instance cause a problem in a change impact analysis.

The procedure for combining slices works in the other direction as well. Listing 3 lists the previously shown example code after the preprocessing phase. Suppose the slicing criterion contains the variable `i` in line 5. The C/C++ backward slice algorithm does not know about macros as the slice only contains lines 4 and 5.

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

Listing 3: Preprocessed example code for combined slices

Using the fact that line 4 comes from a 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.

II/1 Macro slicing

As we already showed, detailed information on macro expansions may be used in two ways. For a macro call, the natural question is to find all the definitions which take part in the full expansion of the macro. This is the most frequently used direction, from calls to definitions. The previously mentioned macro folding mechanism is a good example of this approach. However in the maintenance environment, the other direction may be more important: "Which parts of the program are affected by a change in a macro definition?"

The intuitive method of searching the source tree with a grep-like tool fails for several reasons: (1) includes and configurations, (2) macro re-definitions and (3) hidden macro invocations using ## operators. We presented a formalism to answer similar questions by adapting notions of program slicing.

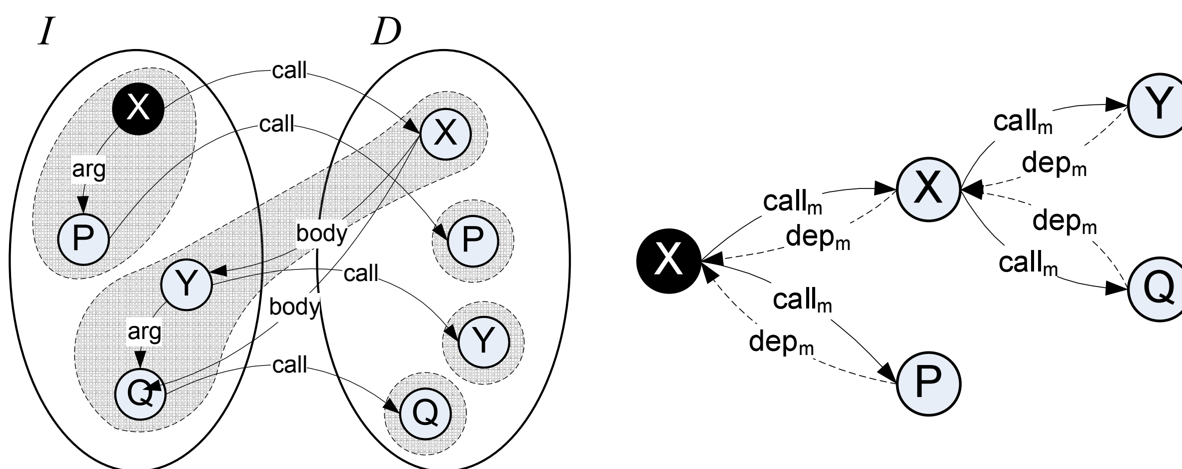

Figure 5: (a) Graph representation of the elements of a macro call, and (b) The $call_m$ and the dep_m relations on the simplified structure

Elements of macros may be represented as graphs. Figure 5 on the left hand side (a) contains an example construct with macro calls, arguments and macro definitions with macro body; and relations among the elements. On the right hand side of the figure (b) the same structure is represented in a compact form: the macro call relation ($call_m$) together with the dependency relation (dep_m) plays an important role in the dependency-based slicing of macros.

The adaptation of traditional slicing concepts required a careful examination of the similarities and differences. With the help of the key definitions, we constructed the Macro Dependence Graph (MDG). A common problem that arises when analyzing macros is the so-called potential macro problem. The result of a macro call is determined by the place of the call and not by the place of the called macro definition. More precisely, any identifier in the macro body may become a macro name at a later point of the program. If the definition of the above-mentioned identifier precedes the call of the original macro, then the original macro body will contain an additional macro call. Therefore one macro definition may be expanded in many ways depending on the place (context) of the call. To overcome these problems and assure the appropriate properties for slicing, dependency edges are colored in the graph. Hence 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. The forward and backward directions also needed reinterpretating. In the dependency-based slicing of C/C++ programs, for example, the direction of the dependency relation is the same as the call relation. 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, with macros the situation is just the opposite: the macro (caller) is dependent on the (called) macro definition. This definition may seem confusing at first, but it is appropriate for linking the two kinds of slices. Finally, we should add that using macro slices, complex macro-related issues may be addressed in a change impact analysis context.

Own contribution

The elaboration of macro dependency-related definitions and the construction of the Macro Dependence Graph are the work of the author. The discussion of the notions of C/C++ and macro slicing, and the definitions of forward and backward slices are shared results. The results of this contribution point were published in research paper [21].

II/2 Combining C/C++ and preprocessor slicing

Here, our novel result is the combination of traditional C/C++ slices with macro slices, giving a more complete dependency set for slicing. The idea of linking the two kinds of slices was already introduced earlier via the motivating example. The dependency-based slicing methods of C/C++ programs and preprocessor macros use a disjunct base set for computing slices. The SDG contains program points, but this notion may have several definitions. A general property of C/C++ program points is that they may overlap each other. Various kinds of program elements are represented by different program points. For instance, line 4 in Listing 3 contains a declaration and an assignment expression as well. A typical SDG does not contain any macro-related program points. However, some program points are the result of macro expansion, which opens up the possibility of combining dependence graphs. the SDG contains the final replacement of the called macro, instead of the macro call. Therefore MDG nodes which represent macro calls may be linked to SDG program points which are derived from the macro replacement (there may be several such program points for a given macro call). In our view, program points are also linked which only partly come from a macro expansion.

We composed a combined dependence graph, which contains both the MDG and the SDG, and the link between them was defined by an extended dependency relation. Technically, the extension for existing dependency relations is based on the source code positions of the macro calls and the replaced texts. Here, the detailed analysis of macro calls is required, which is supplied by the preprocessor schema. An outline of the forward direction of the combining process with macro nodes, program points and the dependency relations is shown in Figure 6. In the forward direction the stating point is a macro definition, then the slice continues via dependent definitions and the MDG part ends in several macro calls. Macro calls are linked to program points, which are the result of the call, and the slice goes on with traditional C/C++ slicing.

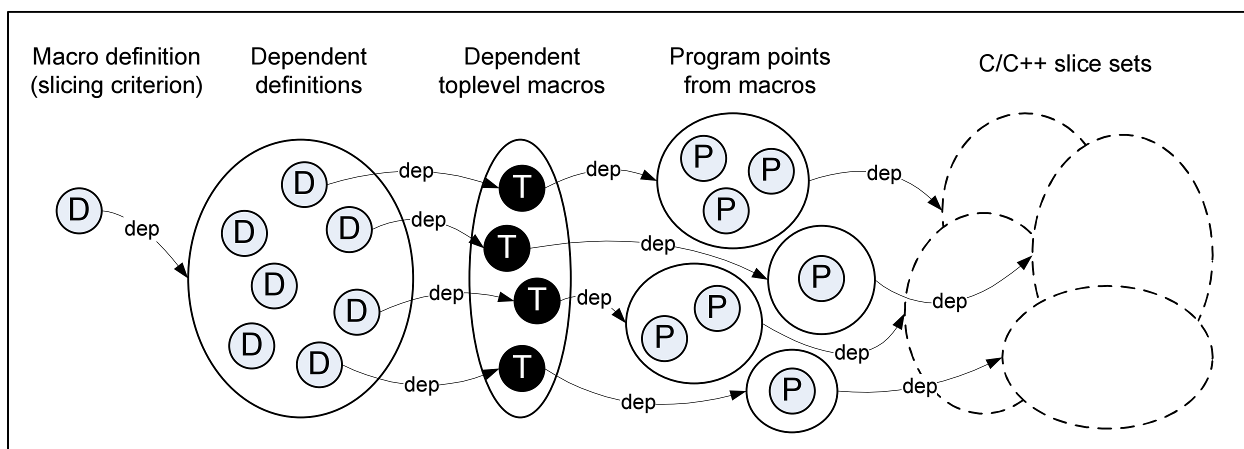

Figure 6: The forward direction for combining the slices, with the dependency relation between macros and C/C++ program points

The combined dependence graph is suitable for computing slices, whose computing process may switch from macro-related nodes to C/C++ program points and vice versa. We gave a formal definition of the combined dependence graph and the combined forward and backward slice sets as well. The combined graph of the motivating example is shown in Figure 7. Graph nodes corresponding to the forward slice starting from the first line, and corresponding to the backward slice starting from the last line are marked with letters `F` and `B`, respectively.

Algorithms for computing both forward and backward combined slices are also given. These algorithms were designed for the global computation of slices, and slightly modified to fit the

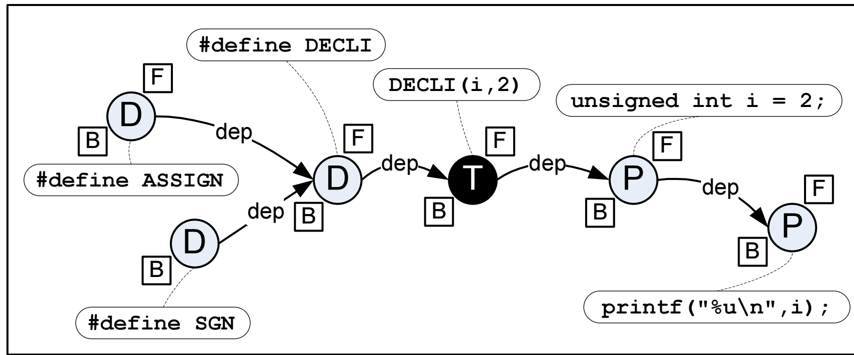

Figure 7: Nodes and slices of the motivating example

tool architecture proposed in the next section. These contributions are significant improvements of traditional C/C++ slicing. In the forward case the improvement is more apparent because C/C++ slicing would miss even the slicing criterion itself.

Own contribution

The construction of the combined dependence graph and the slicing algorithms are the work of the author. The definitions of combined forward and backward slices are shared results. The results of this contribution point were published in research papers [24, 22]. Our work was honored with the *Best Paper Award* of the 16th IEEE International Conference on Program Comprehension in 2008.

II/3 Experimental evaluation of slicing methods

Besides the theoretical results related to macro slicing and combining, we performed experiments to evaluate the outcome of our proposed methods. These experiments were performed both in macro slicing and combined slicing areas.

Macro slices

The macro slicer tool is implemented on the top of schema instances, since schema instances contain all the information necessary for slicing, and they play the role of the Macro Dependence Graph. As schema instances can be produced from large, industrial-size software, our question was whether macro slicing could be done on a large scale. A more important aspect was that, although a thorough empirical study on the preprocessor provided clues on macro use [6], before our study the size of the slices and the distribution of small and large slices could only be roughly estimated. Our experiments were performed on the source code of Mozilla Firefox – which would be a hard task for a C/C++ slicer.

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

Table 1: Summary of macro definitions and expansions

The number of macro definitions and full expansions found in the source code are listed in Table 1. The number of macro calls is high, there being 90 macro definitions which are called over 1000 times. Table 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 2: Summary of macro calls and slice sizes

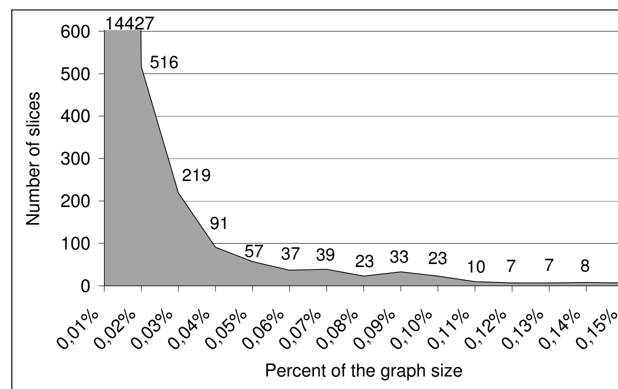

Figure 8: Histogram of slice sizes relative to the graph size

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 8 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 work 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.

Combined slices

We also outlined a tool architecture needed to implement a preprocessor-aware C/C++ slicer, which computes combined slices. 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.

Experiments were performed on 28 open source projects, starting from small programs to medium-sized ones with about 20k lines of code. Many of the programs were selected based on

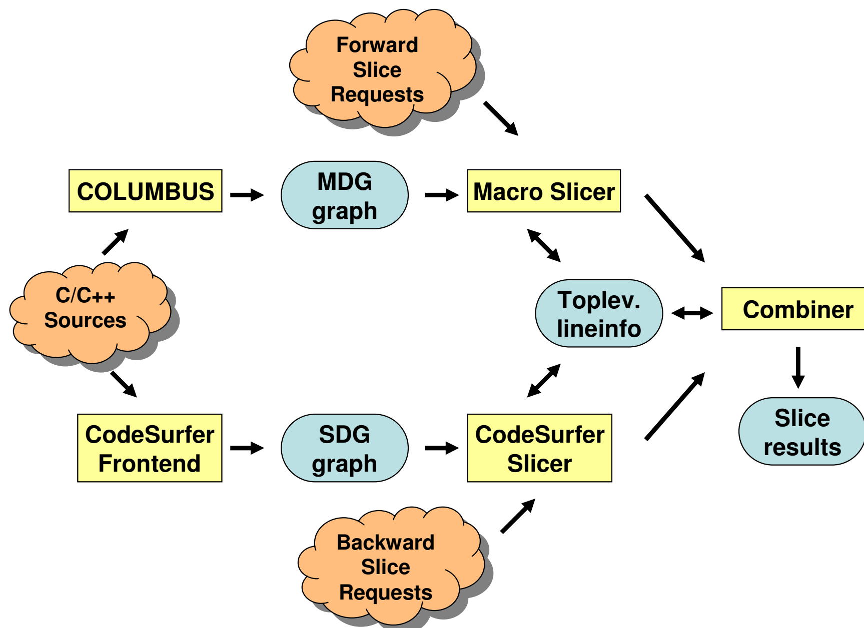

Figure 9: Logical tool architecture for combined forward and backward slicing

comprehensive empirical studies on slicing [2] and preprocessor use [6]. We found a total of 240k non-empty lines of code sufficient to justify the feasibility of the method. Table 3 contains a list of projects used in our measurements and their basic statistics.

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.

Own contribution

The results relating to the evaluation of macro slicing are the work of the author (macro slicing using schema instances, the implementation of the macro slicer, the experimental evaluation of macro slices). The results relating to the evaluation of combined slicing are the work of the author (tool architecture for combining slicing, the implementation of the combined slicer, the evaluation of C/C++ and macro slices), except for the shared work on implementing the CodeSurfer plugin. The key results of this contribution point were published in research paper [22].

Conclusions

Our research over the past few years has been dedicated to the support of program maintenance activities impeded by the presence of preprocessor directives. Recognizing that after several years of development and operation, the source code is the only relevant and complete documentation of a program, we applied a source code-based reverse engineering approach. We presented 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 based on the programs being analyzed. The tool is called CANPP, and is capable of analyzing industrial sized software projects with millions of lines of code. The preprocessor

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 3: Subject programs

schema, and the API that provides access to schema instances, together allow the use of detailed information for further analysis purposes in program comprehension, like macro folding or for investigating the include hierarchy. Inter-operation 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. Refactoring 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 and implemented a tool architecture, 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 the slicing notions introduced in our work were validated by experiments. The schema instances served as the MDG, and our macro slicer tool was 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 an area where a number of configuration-related issues are waiting to be addressed. 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.

The relation between the contribution points and the supporting publications, which are all first-author papers, can be seen in Table 4 below. We introduced novel methods to assist the maintenance tasks of preprocessed languages. Both theoretical and practical results were achieved, and several tools were implemented. Experiments were then performed to demonstrate the practical utility of theoretical results in the areas of modelling, refactoring and slicing preprocessed languages.

	Contribution - short title	Publications
I/1	Preprocessor metamodel	[19] [20]
I/2	Model level refactoring	[23] [18]
II/1	Macro slicing	[21]
II/2	Combining slicing	[24] [22]
II/3	Evaluation of slicing methods	[22]

Table 4: Thesis contributions and supporting publications

Acknowledgements

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.

László Vidács, October 2009.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

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

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

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

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

[11] 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.

[12] 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.

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

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

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

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

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

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.