MACRO IMPACT ANALYSIS USING MACRO SLICING

László Vidács, Árpád Beszédes and Rudolf Ferenc
Department of Software Engineering, University of Szeged, Hungary
lac@inf.u-szeged.hu, beszedes@inf.u-szeged.hu, ferenc@inf.u-szeged.hu

Keywords: Change impact analysis, macros, preprocessing, C, C++, program understanding, program analysis, maintenance, program slicing, dynamic analysis.

Abstract: The expressiveness of the C/C++ preprocessing facility enables the development of highly configurable source code. However, the usage of language constructs like *macros* also bears the potential of resulting in highly incomprehensible and unmaintainable code, which is due to the flexibility and the "cryptic" nature of the preprocessor language. This could be overcome if suitable analysis tools were available for preprocessor-related issues, however, this is not the case (for instance, none of the modern Integrated Development Environments provides features to efficiently analyze and browse macro usage). A conspicuous problem in software maintenance is the correct (safe and efficient) management of change. In particular, due to the aforementioned reasons, determining efficiently the impact of a change in a specific macro definition is not yet possible. In this paper, we describe a method for the impact analysis of macro definitions, which significantly differs from the previous approaches. We reveal and analyze the dependencies among macro-related program points using the so-called *macro slices*.

1 INTRODUCTION

C/C++ source code analyzer tools many times 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 coping with the preprocessor constructs, only partial and imprecise results can be obtained. Lots of efforts are already put into incorporating the preprocessor related information into the processes which analyze the C/C++ language constructs but only with 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++ face many problems because of preprocessor constructs (Garrido, 2005).

In this paper, we concentrate on understanding macro usage. Usually, macro related analysis is used to track the macro call to its definition. Although research tools implementing this feature (e.g. the folding mechanism of GUPRO (Ebert et al., 2002)) already exist, the widely used debuggers still do not provide this information. 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 the one above, 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 changes without having tool support for analyzing the impact of the change to the code, which may cause unforeseeable 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 (Rajlich, 1997). In particular, when analyzing the impact of changes in macros, we need to know all usages of a macro definition. In other words, it is needed to track the macro definition to all of its usages (macro calls) – as opposed to the other direction mentioned previously.

Our motivating question is hence: *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 intu-

itive method is to search the whole source tree using the *grep* tool to find all occurrences of the name of the modified macro definition. Unfortunately, there are three main obstacles which make this method unusable: includes and configurations, macro redefinitions and hidden macro invocations using the ## operator.

In this paper, we introduce a *novel technique* which answers the motivating question. The next section contains the necessary terms and definitions for the analysis of macros. In Section 3, the macro slicing method is introduced. Related research is discussed in Section 4. The last section contains conclusions and closing remarks.

2 DEFINITIONS

When investigating preprocessor directives, the meaning of static and dynamic analysis is different than the usual. The preprocessing phase takes place before the compilation, configurations of the program are controlled by an initial set of macros. 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 considering 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 choose the 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 an important 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 1, the macro call results in `1 2`):

- *macro definition* – the place of the `#define` directive. The definition consists of three parts: *macro name*, optionally *parameters*, and *macro body* (also called 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). The invocation may contain *macro arguments* in case of function like macros.

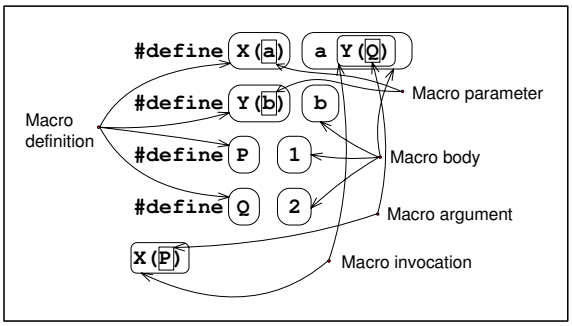

Figure 1: Example macro call.

- *macro expansion* – the process of macro replacement: macro arguments are expanded and replaced.
- *full macro expansion* - starting from the point of a macro invocation there may be many expanded macros since the macro body may contain further macro invocations. On full macro expansion we mean all expansions which are necessary to get the final result of the beginning macro invocation.
- *toplevel macro invocation* - starting point of a full macro invocation (a full macro expansion necessarily starts outside the `#define` directives).

Definition Let I be the set of all macro invocations in the given software.
Definition Let D be the set of all *used* macro definitions in the given software.

The fact of a macro call is represented by the *call* relation between the two sets.
Definition $call : I \to D$, $call(x) = y$ 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 is not injective (one definition can be called from more places).

A macro invocation may contain arguments in case of function like macros. These arguments may also contain macro invocations, so we define the following relation.
Definition $arg : I \to I$, $arg(x) = y$ if and only if the macro invocation x calls a function-like macro and the macro invocation y is an argument of x.

A macro definition may contain further macro invocations in its body. This relationship is represented by the following relation.
Definition $body : D \to I$, $body(x) = y$ 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 later 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* re-

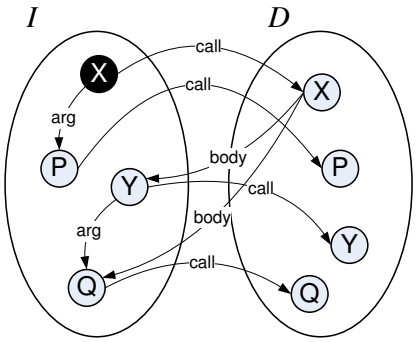

Figure 2: Macro sets and relations.

lations (similarly to a graph edge contraction). Let us construct a new set called *MC* containing disjunct node sets containing elements from *I* and *D*. There are two types of new nodes. The first type is based on toplevel macro invocations (filled with black in Figure 2): each set contains a toplevel invocation and invocations which are in its arguments (contraction using the *arg* relation). The second type is based on macro definitions: each set contains a macro definition and macro invocations contained by its macro body (contraction using the *body* relation). In Figure 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:

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

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

$$MC = MCI \cup MCD$$

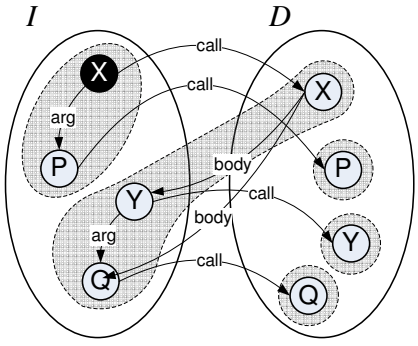

Figure 3: Elements of the *MC* set.

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 by one of the elements of *MC*. The *call* relation can be defined on MC as follows:

Definition $mcall : MC \rightarrow MC$,
$mcall(A) = \{B | \exists x \in A, y \in B : call(x) = y\}$.

Macro dependencies can be defined based on the *mcall* relation. Note that the dependency edge points to the opposite direction than the *mcall* edge.

Definition $dep : MC \rightarrow MC$, $dep(a) = b$ if and only if $mcall(b) = a$.

Figure 4 shows the simplified set. Node sets (filled areas in Figure 3) are represented by their base nodes in Figure 4.

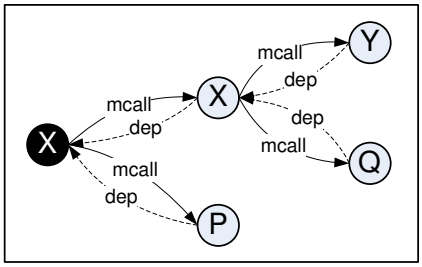

Figure 4: The *mcall* and the *dep* relations on the simplified *MC* set.

3 SLICING

Program slicing is an analysis method for extracting parts of a program which represent a specific sub–computation of interest. It has been originally introduced by Weiser (Weiser, 1984) 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 *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 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 needs only to preserve meaning for the input under consideration.

Over the years, a number of algorithms to compute program slices has been developed; for an overview see (Tip, 1995; Xu et al., 2005). One of the most cited approaches is to apply a pre-computation step in which a representation of the program under investigation is constructed first, which captures the *dependences* among program elements (for instance,

data dependences). 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.* (Horwitz et al., 1990). The nodes of this graph represent the program elements (instructions), while the edges connecting them correspond to the program dependences. The counterpart of this graph for dynamic slicing, the Dynamic Dependence Graph (Agrawal and Horgan, 1990) 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). Eventually, 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, 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 dependences (Beszédes et al., 2006). 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 dependences are investigated in order to determine a particular program slice.

In this work we reuse the basic slicing principles to compute *macro slices*. Namely, we construct the Macro Dependency Graph (MDG), with which *forward dynamic macro slices* can be computed, which will serve as a solution to our initial problem of analyzing impacts of changes in a macro definition. However, as we will see in the following, a number of slicing concepts need to be reinterpreted in the scope of macro slicing.

3.1 Macro Slicing

Using the approach which restricts the slice criteria to used and defined variables we 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.

Definition 1 *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 2 *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 $< p,x >$ gives the answer to the motivating question outlined in the introduction.

Slices can be produced based on the *mcall* and *dep* relations using the definitions from Section 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 *mcall* and *dep* 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 have to be refined.

The problem is 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 5, at the point of the definition of macro *X* identifier *Y* is a simple identifier, but it becomes a defined macro later. At the point of the second invocation of macro X the identifier Y is a macro, so the full expansion of macro X staring from that point contains the expansion of macro Y.

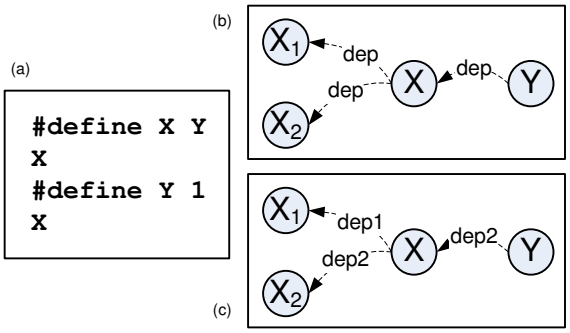

Figure 5: Potential macro problem: (a) program code (b) basic graph (c) MDG with edge coloring.

The question is which points are affected when the definition of Y is modified. A search based on the *dep* relation starting from the macro definition *Y* finally finds both *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 track back macro replacements separately.

After the preparations let us construct the Macro Dependency Graph (MDG). The nodes of the graph are the elements of the *MC* set and the directed edges are created from the *dep* 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 sign the edges that belong to a particular full macro expansion.

Definition 3 *Let $MDG = (V,E,I,C)$ be the Macro*

Dependency Graph, where V is the set of nodes (vertices) and E is the set of edges, $I \subseteq V \times E$ is the incidence relation, for $\forall e \in E$ the $\{e \in V : vIe\}$ set has two ordered elements (the endpoints of the edge), and $C \subseteq E \times \mathbb{N}$ is the coloring relation which assigns the same color to the edges which belong to the same full macro expansion. The set E contains multiple edges colored with different colors, if more full expansions would use the same edge.

Producing slices can be done on the MDG. For a slicing criterion $< p,x >$ there is a node $k \in MC$ in the dependency 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 4 *Let $< p,x >$ be a slicing criterion where x is a definition and $k \in MC$ the node corresponding to x. Let Col be the set of colors which are used on dependency edges starting from k:*
$Col = \{c \in \mathbb{N} | c \in C(e) : e \in E, \exists l \in V : (k,l) \in I\}$.
The forward macro slice of the criterion is the set $S = \{y \in MC | y \in dep_i^t(k), i \in Col\}$, where dep_i^t is the transitive closure of dep colored with i.

Because of edge coloring the search process of the slice elements is modified: starting from the criterion only those elements belong to the slice which are reachable through edges colored by those colors which start from the criterion node. An example graph can be found in Figure 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, usually 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 have to be merged. Merging dependencies – in extreme cases – may bring cycles into the graph. To overcome this problem each merged source file has to 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 *mcall* relation are added with the appropriate coloring. Let I^{dep}, E^{dep} and I^{mcall}, E^{mcall} be the set of edges and incidence relations based on the *dep* and *mcall* relations respectively. Let $MDG = (V,E,I,C)$ where $E = E^{dep} \cup E^{mcall}$ and $I = I^{dep} \cup I^{mcall}$. The forward slice is computed on *dep* edges while the backward slice is computed on *mcall* edges.

[1] According to the preprocessor standard, if a macro is under expansion and during the re-expansion the same macro is called again, then further calls will not take place (the macro name remains in the replacement list instead).

3.2 Discussion on Macro and Procedural Slices

In their first approach, Agrawal and Horgan introduced dynamic slicing by refining the static Program Dependence Graph using information from the execution history (Agrawal and Horgan, 1990). The need for the Dynamic Dependence Graph to construct 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 execution history. In the case of macro slicing the set of *mcall* 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 invocation, 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.

For computing macro impacts we determine macro slices that we refer to as *forward slices*. It is interesting to observe that the 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. However, in the case of macro programs, the notion of "order" is less obvious since there are no "executable instructions" either (consider, for example, that the macro dependency edge points in the reverse direction as the macro call edge, while with procedural programs the control flow aligns with the control dependency). Furthermore, it is meaningless to talk about data dependencies too in the case of macro slicing, since these may exist only between the actual arguments and the formal parameters, however the macro definition itself is not a part of the program, and therefore the data dependency starts from the point of the initial call and necessarily ends at the same place.

4 RELATED WORK

The usefulness of the preprocessor is proved by many years of use by developers. The opinion is the opposite when one has to aid maintenance or program understanding tasks: the presence of preprocessor directives is always mentioned as an obstacle (Spencer and Collyer, 1992). Therefore, lots of efforts were made to avoid their usage. Mennie and Clarke proposed a method to transform some macros and condi-

tionals into C/C++ code (Mennie and Clarke, 2004). Spinellis tackles the problem of global renaming of variables (Spinellis, 2003).

There are remarkable contributions which offer a solution to the opposite direction of our question: when seeing a macro name in the source code, which macro definitions take part in the expansion. The GUPRO program understanding framework (Ebert et al., 2002) implements a macro folding mechanism: a macro can be hidden or revealed at the place of the call (Kullbach and Riediger, 2001). Livadas and Small identify mappings between the preprocessed and the unprocessed code. The approach is implemented in the GHINSU software maintenance environment, where by clicking on a macro invocation, the called definitions are highlighted (backward macro slice using our terms) (Livadas and Small, 1994). A flexible solution is offered by Badros and Notkin: the PCp^3 C analysis tool defines callback perl functions for preprocessor activities (Badros and Notkin, 2000). These methods require only the analysis of the compilation units. In our approach, however, we need to use information from the whole source. (Note that by using our approach backward slices can also be computed.)

The Understand for C++ reverse engineering tool provides cross references between the use and definition of software entities (Understand for C++ Homepage, 2007). This includes the step-by-step tracing of macro calls in both directions. The user can track back the usages of a given macro definition easily but the information is not accurate. The program fails on the problem shown in Figure 5 and, for example, it misses calls using ## or shows a macro call where a parameterized macro name is used without arguments, so no macro expansion happens.

5 CONCLUSIONS

As a response to the lack of complete solution to the macro change impact problem, we introduced an approach based on macro slices. Based on the relations between macro invocations and definitions, we construct a Macro Dependency Graph on which macro slices can be computed. By using multiple edges and edge coloring, this graph handles potential (and later defined) macro names and macro re-definitions.

As a proof of concept, an experimental tool based on the Columbus C/C++ frontend (FrontEndART Homepage, 2007) has been developed. We have already performed some preliminary experiments which proved our concepts. In the future we plan to evaluate the method in some more detailed case studies. We also plan to produce backward macro slices and to implement an efficient algorithm for global computation of macro slices (and not demand driven as the current one).

REFERENCES

Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *Proceedings of the ACM PLDI 1990*, pages 246–256, New York, NY, USA. ACM Press.

Badros, G. J. and Notkin, D. (2000). A Framework for Preprocessor-Aware C Source Code Analyses. *Softw. Pract. Exper.*, 30(8):907–924.

Beszédes, Á., Gergely, T., and Gyimóthy, T. (2006). Graphless dynamic dependence-based dynamic slicing algorithms. In *Proceedings of SCAM 2006)*, pages 21–30.

Ebert, J., Kullbach, B., Riediger, V., and Winter, A. (2002). GUPRO - Generic Understanding of Programs. In *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier.

FrontEndART Homepage (2007). `http://www.frontendart.com`.

Garrido, A. (2005). Program refactoring in the presence of preprocessor directives. Ph.D. thesis, UIUC.

Horwitz, S., Reps, T., and Binkley, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61.

Kullbach, B. and Riediger, V. (2001). Folding: An approach to enable program understanding of preprocessed languages. In *Proceedings of WCRE 2001*, pages 3–12, Los Alamitos. IEEE Computer Society.

Livadas, P. E. and Small, D. T. (1994). Understanding code containing preprocessor constructs. In *Proceedings of IWPC 1994*, pages 89–97. IEEE Computer Society.

Mennie, C. A. and Clarke, C. L. A. (2004). Giving meaning to macros. In *Proceedings of IWPC 2004*, page 79, Washington, DC, USA. IEEE Computer Society.

Rajlich, V. (1997). A model for change propagation based on graph rewriting. In *Proceedings of ICSM 1997)*, pages 84–91.

Spencer, H. and Collyer, G. (1992). #ifdef considered harmful, or portability experience with C News. In *Technical Conference*, pages 185–197.

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

Tip, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189.

Understand for C++ Homepage (2007). `http://www.scitools.com`.

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

Xu, B., Qian, J., Zhang, X., Wu, Z., and Chen, L. (2005). A brief survey of program slicing. *ACM SIGSOFT Softw. Eng. Notes*, 30(2):1–36.