Combining Preprocessor Slicing with C/C++ Language Slicing

László Vidács, Judit Jász, Árpád Beszédes and Tibor Gyimóthy
University of Szeged, Department of Software Engineering
Árpád tér 2., H-6720 Szeged, Hungary, +36 62 544143
{lac,jasy,beszedes,gyimi}@inf.u-szeged.hu

Abstract

Slicing C programs has been one of the most popular ways for the implementation of slicing algorithms; out of the very few practical implementations that exist many deal with this programming language. Yet, preprocessor related issues have been addressed very marginally by these slicers, despite the fact that ignoring (or handling poorly) these constructs may lead to serious inaccuracies in the slicing results and hence in the comprehension process. Recently, an accurate slicing method for preprocessor related constructs has been proposed which – when combined with existing C/C++ language slicers – can provide a more complete comprehension of these languages. In this paper, we overview our approach for this combination and report its benefits in terms of the completeness of the resulting slices.

Keywords

Program slicing, C/C++, preprocessing, preprocessor slicing.

1. Introduction

There is a seemingly small, but important factor which differentiates C and C++ from other programming languages from the program understanding point of view: the preprocessor. It is not strictly part of the C/C++ language syntax, but it cannot be separated from it. The absence of the constructs and the possibilities provided by the preprocessor would make programming virtually impossible with these two languages.

The two forms of source code (original and preprocessed form) are always mentioned as an obstacle in program comprehension. 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. Researchers in some fields need to cope with the preprocessor, for example a refactoring transformation cannot be performed safely without preprocessor analysis [24, 25]. Even a transformation as simple as a "rename variable" requires the analysis of preprocessor configurations [10].

Program slicing [19, 26] is seen as a very powerful technique applicable in various fields related to program comprehension and maintenance in general. These include specifically: change impact analysis [4], program decomposition [9], software re-use [5, 27], debugging [1] and regression testing [3, 18]. However, preprocessor issues are many times completely neglected by slicing algorithms, or at least, handled very poorly. Features like file inclusion or conditional compilation are sometimes handled in an acceptable way, but macro expansion is a different story. The best what existing slicers do about them is to mark those program points coming from macros and display this information to the user. CodeSurfer [11] for instance – which is known as the probably best static C/C++ slicer available as of today –, displays information on macros appearing in slices, but is unable to involve them in slicing itself. A remarkable exception is the Ghinsu C slicing tool [14] which implements notable features for comprehending programs with preprocessor constructs, but unfortunately this project seems not be maintained anymore.

In recent work, the notion of *macro slicing* has been introduced [23]. In this approach slices can be computed on the structure of preprocessor macro calls and macro definitions. This is enabled 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 many definitions. In the opposite direction the text of a macro definition may be used (through other definitions) in many macro calls. Separating the relevant replacements in a specific macro usage is the task of macro slicing (in the following, we refer to this kind of slices as *macro slices*).

However, 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 existing 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 slices* is the following). The combined slice contains more accurate information about the C/C++ program as we will see in the remaining of the paper, which is very important from program comprehension point of view. In this connection a special role is played by the toplevel macro calls (which are in program text, not in 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. (See below for a motivating example.)

In this paper, we discuss the method and the issues of connecting slices, and report the results of our first experiments. Next section contains a motivating example, then we introduce macro slices and the connection of the two kinds of slices in Section 3. Section 4 reports the empirical results of our implementation, while Section 5 contains a discussion about related research activities. Finally, in Section 6 we summarize our contribution.

2. Motivating example

Macro slices can be used, among others, for change impact analysis purposes. The developer usually has to carry out small changes during the system maintenance tasks, but in a large software the effect even of a small change is hard to predict. Let us assume that the small program part to be changed is a macro definition. Our first motivating problem is to find the points of a C/C++ program which are affected by a changed macro definition. The changed definition may be used in (called from) other macro definitions, which can be called again from many points of the program (these are conceivable as macro slices). Finally, 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 language slice starting from replaced parts gives all the affected points. A small example illustrating this issue can be seen in Figure 1.

The slicing criterion for macro slicing is the macro definition at line 1. The corresponding macro slice contains lines 1, 3 and 4, while the macro call at line 4 is the connection between the two kinds of slices. During preprocessing,

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

Figure 1. Small example on macro and regular forward slices (note that there is a line break in line 3 to fit in width)

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 at line 1 affects four lines. Failure to identify these additional dependencies can be a problem in change impact analysis, for example.

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

Figure 2. The example source code after preprocessing

The combination of slices works in the opposite direction as well. Figure 2 shows the example code after the preprocessing phase. 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, the slice contains lines 4 and 5. 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 all lines of the original example, instead of two lines of the C/C++ slice. An example where this can cause problems is that if this additional information is not available in a debugger, the user could not track down to all possible causes of a failure debugged.

In the next section we overview our approach to compute such combined slices.

3. Combining C/C++ preprocessor and language slices

The connection between the two types of slices can be performed in both 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 are (part of) C/C++ program elements, which 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.

In this section, we first overview macro slices and then we describe our approach for combining the two kinds of slices.

3.1. Macro slices

For a detailed description of macro slices we refer to our previous work [23]. Here, only an overview is presented with some figures and definitions.

Slices are usually defined on a graph structure which represents dependency relations between program elements. Accordingly, the structure of macros is defined by using sets and relations, and a dependency graph is defined based on macros using the dependency relation which is appropriate for slicing macros.

The following terms are used to formalize the macro replacements (see the example in Figure 3, the macro call results in `1 2`):

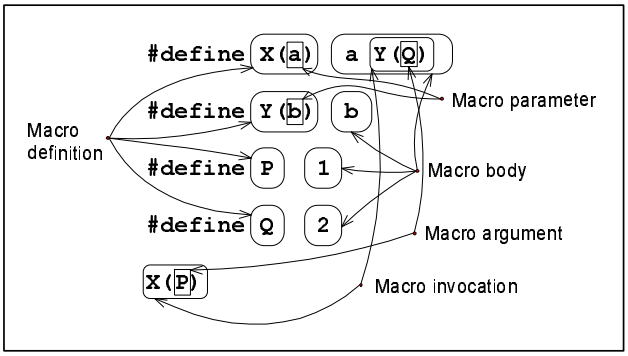

Figure 3. Example macro call

- *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 the case of function like macros because there may be more macro invocations in the same line with the same name.

- *macro expansion* – the process of macro replacement: macro arguments are also expanded and replaced.

- *full macro expansion* - the 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 the 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).

Let us construct a set called MC containing macro invocation nodes and macro definition nodes. Both types of nodes are multi nodes (node sets) in the sense that they contain many preprocessor elements, but for the sake of simplicity and readability we use them as one node. The first type is based on toplevel macro invocations (filled with black in Figure 4): each node contains a toplevel invocation and the invocations which are in its arguments. The second type is based on macro definitions: each node contains a macro definition and the macro invocations contained by its macro body. The set is constructed in order to eliminate all relations other than macro calls. Based on macro calls, the dependency relation on the MC set can be defined as follows (the $mcall(x) = y$ means that macro x calls definition y):

Definition 1 *$dep : MC \rightarrow MC$, $dep(a) = b$ if and only if $mcall(b) = a$.*

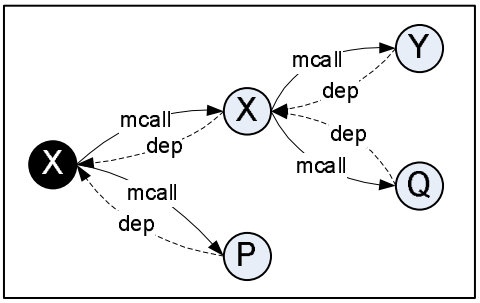

Figure 4. The $mcall$ and the dep relations on the simplified MC set

An example set with relations can be seen in Figure 4. The dependency arrow points to the opposite direction than

the $mcall$ arrow. Informally a node is dependent from another if and only if there is a macro call from inside the first node to the second node.

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 2 *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.*

It is important to note that the MDG is an acyclic graph even when it contains subgraphs of the whole software system.

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 3 *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.

Backward macro slices can be defined similarly, the slice starts at a macro call and contains all definitions which are used during the full expansion of the macro.

A small example source code and dependency graph can be found in Figure 5. The dependency edge colors are shown as numbers. The forward macro slice based on the definition of Y as a criterion contains the definition of X and the second macro invocation X_2.

3.2. Connecting slices

The combination of macro and language slices requires a common set of nodes and edges to be defined with the dependency relation as well. C/C++ language slices are usually computed on some kind of a Program Dependence Graph (PDG) [16], or more generally on a System Dependence Graph (SGD) introduced by Horwitz *et al* [12]. The SDG models interprocedural dependencies between procedures where each procedure is modelled with a PDG. The MDG can be constructed to contain dependencies from all compilation units in a software, there is no need to define two kinds of graphs for the macros.

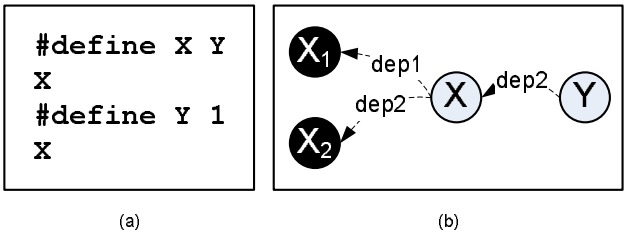

Figure 5. Example code and MDG: (a) program code (b) MDG with edge coloring

The MDG can be used in combination with the SDG as follows. Both of them have a well defined structure, the only problematic point is the connection. The MDG is based on the original source code, while the SDG contains C/C++ language elements. Practically 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 2.). The macro call is replaced with the replacement text. From that point the source code is really in C/C++ language and consists of C/C++ program elements.

Unfortunately, there is no obligation for the replacement text to be a C/C++ syntactical unit. Moreover, the SDG is built up of program elements, but contains various kinds of nodes like declaration, expression, return and so on. The macro replacement may be a sequence of statements which is represented by more nodes in the SDG, and the macro may even be a constant which is only a part of an SDG node. This means that there is a many-to-many relation between macro replacement texts and SDG nodes. Some kind of dependency relation can be defined between the SDG nodes that at least partially come from a macro replacement and between the macro call. Let $repl(a)$ be the replacement text 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 4 *Let $dep_{comb} : MDG \rightarrow SDG$, $a \in MDG$, $b \in SGD$, $dep(a) = b$ if and only if a is toplevel, $\exists x : x \in repl(a) \wedge x \in b$.*

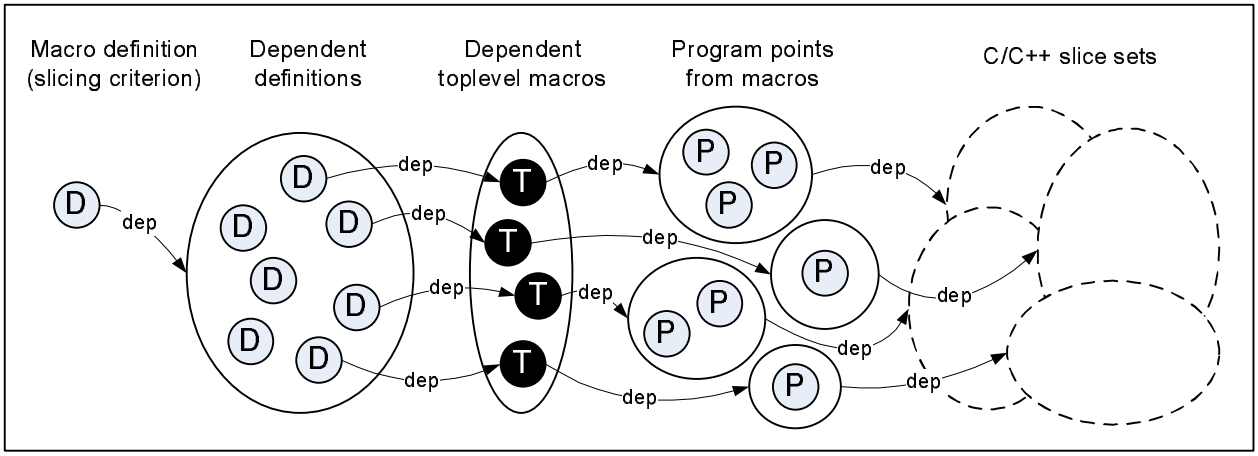

Figure 6. Forward direction of combining slices - dependency relation between macros and C/C++ program points

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 existing definitions, the combined slice can be defined. Let dep_m be the macro dependency and dep_{cc} be the C/C++ dependency relation. Let CombDG be the combined dependency graph and dep the combined dependency relation:

$$CombDG = SDG \cup MDG$$

$$dep(x) = \begin{cases} dep_m(x), \textit{if } x \in MDG \\ dep_{cc}(x) \cup dep_{comb}(x), \textit{if } x \in SDG \end{cases}$$

Definition 5 *Let $< p, x >$ be a slicing criterion of program p, where $x \in p$ is a program point. The combined forward slice of the criterion is the set $S = \{y \in p | y \in dep^t(x)\}$, where dep^t is the transitive closure of the dep relation.*

Definition 6 *Let $< p, x >$ be a slicing criterion of program p, where $x \in p$ is a program point. The combined backward slice of the criterion is the set $S = \{y \in p | x \in dep^t(y)\}$, where dep^t is the transitive closure of the dep relation.*

The forward direction is depicted in Figure 6. The capital letters in figure elements reflect their type in this figure (and not their names). 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 there are many dependency edges among the elements of this set omitted from the figure.) When toplevel invocations are replaced, the result of each invocation takes part in a set of C/C++ program elements (P). A regular language slicing algorithm computes the slice for each program element, so the final combined slice contains all elements in the figure.

The backward direction is outlined in Figure 7. Here also, the capital letters in figure elements reflect their type and not their names. The slicing criterion is a C/C++ program element (P). The slice may contain SDG nodes which are (at least partially) the results of one or more macro invocations. The toplevel invocations which take part in the C/C++ slice can be found along the dependency edges. For all of these toplevel invocations, macro slice sets can be obtained using backward macro slicing. The final combined backward slice contains all elements in the figure.

The combined graph and the combined slices of the sample source code from Section 2 can be seen in Figure 8. Nodes belonging to forward and backward slices are denoted with a capital 'F' and 'B' respectively. The toplevel macro call `DECLI(i,2)` is present in both of its forms: as a macro and as a program point. The forward slice contains all nodes but the definition of `SGN`, while the backward slice contains all nodes of the graph.

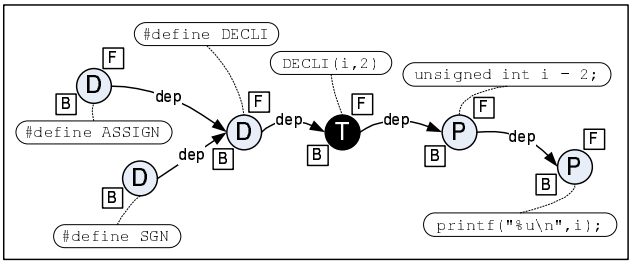

Figure 8. Nodes and slices of the motivating example

Note that the method does not use special information about the SDG of the C/C++ slicing algorithm. We only use the dependency relation and the character positions of the node texts. Therefore, in theory the method can be used

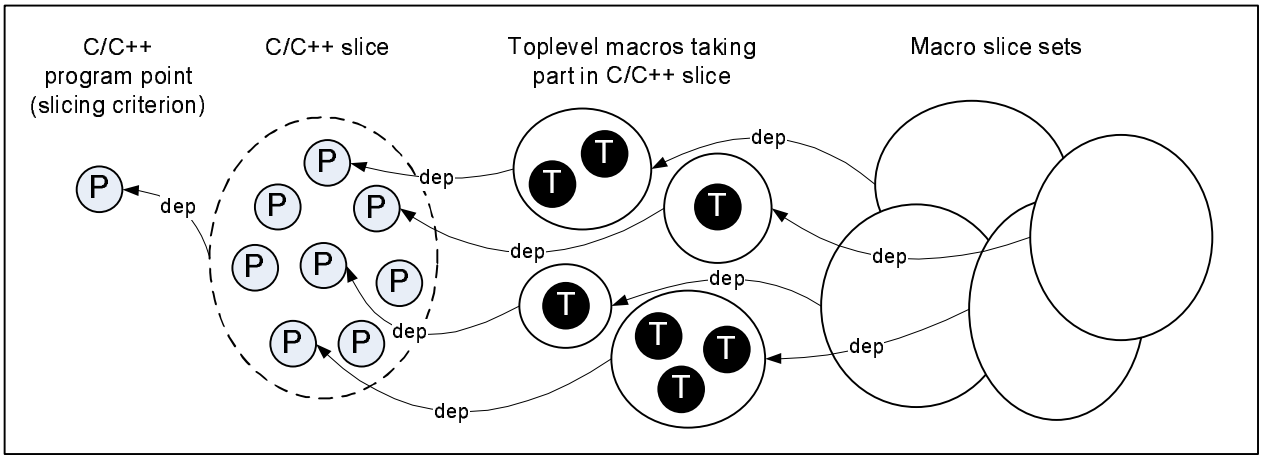

Figure 7. Backward direction of combining slices - dependency relation between macros and C/C++ program points

for static or dynamic slicing, furthermore, it does not matter whether data, control or other dependency relation is used for slicing.

4. Measurements

4.1. Tool setup

We have set up an experimental toolchain for computing combined slices in which we mainly reused and integrated existing tools. From the preprocessor part we added new features to our macro slicing tool, and from the C/C++ part we implemented a CodeSurfer plugin to get slicing information [11]. The macro slicer is built on top of the Columbus technology [7, 8]. In the first stage, the tool analyses the project and, as a result, creates a graph instance of the preprocessing schema [22]. The graph contains dependency edges between preprocessor elements on which macro slicing can be done. Similarly, CodeSurfer builds the SDG graph representation from a software project and determines language level dependencies (among many other information). CodeSurfer gives access to the internal representation of the SDG and the dependency information through plugins. We used the C API, which is appropriate for using the core functionality (the Scheme API provides full access).

The outline of the toolchaing can be seen in Figure 9. It consists of the macro slicer tool, the CodeSurfer plugin and a small tool which summarizes the results. The tools communicate with each other through a set of toplevel macros, which are presented by their line information, which is the common denominator of the two slicers.

In the case of forward slicing the process starts in a macro definition as the slicing criterion. The macro slicer produces the macro slice whose final result is the set of toplevel macros, which is provided to the language slicer through its plugin. In the next step CodeSurfer identifies places in the source where macro replacement was done, which is available by line and column information in the vertices returned. The match between toplevel macros and vertices is done based on this information (from the various types of vertices only those are used which have position in the source). Finally, the language slicing algorithm is executed to produce slices for each toplevel macro. The results are summarized for each beginning macro definition criterion (the C/C++ slice is the union of the slices belonging to the toplevel macros).

In the backward direction we start with a C/C++ program point as the criterion. For it, the CodeSurfer plugin produces the backward slice, which is then scanned for vertices which are results of a (toplevel) macro call. The slice is written into the output, and the set of toplevel macros contained by the slice is given to the macro slicer tool. The macro slicer counts backward slice sets for each toplevel macro and this way it extends the existing slice. Finally the slices are summarized.

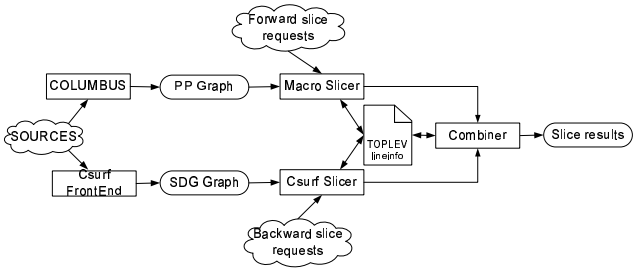

Figure 9. Outline of the tools

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: physical and logical lines (for example in the case of `#line` directive CodeSurfer preserves the original line information), handling macros in conditional directives, and handling macros defined in command line. The CodeSurfer plugin iterates through vertices belonging to procedures, which means that some vertices are omitted (forward declarations, for example). Another important factor is the handling of standard libraries. The SDG contains some additional vertices from standard libraries (not all of them), and some vertices used in its internal representation. Accordingly, the macro slicing tool is adjusted to match macros from standard libraries, but not to complain about omitted ones.

4.2. Subject programs

We performed the experiments on some small open source projects. There is a wide range of open source software which are analyzed by Ernst et al [6]. They report the preprocessor directive usage in open source software and find that *flex* can be treated as average in terms of preprocessor directive usage (preprocessor directives make about 8.4% of program code). Therefore we also chose this program as the main subject for our experiments. It is relatively small but a relevant project in this context. The projects used in our measurements with their sizes can be seen in Table 1 (sizes given in non empty lines of code and node numbers, respectively).

	Size (neLOC)	MDG size (nodes)	SDG size (nodes)
flex-2.5.34	18346	1781	131052
time-1.7	1975	162	5633
wdiff-0.5	3283	217	7643
ed-0.8	2662	117	38756

Table 1. Subject programs

4.3. Slices in details

In our experiments we used number of source code lines which contain vertices from the slice, since this is the best common denominator for the different slicing tools. Other researchers also followed this approach [2].

Because of the difficulties in matching mentioned in the previous section, there were slices in both directions which the tools failed to match. The failure rate was generally about 8% in forward case, and less than 0.1% in backward case, which we found acceptable for the first experiments. The data given in this section contains only the perfectly matched slices.

The number of combined forward slices and their average sizes can be observed in Table 2. We computed all possible forward slices, meaning that we started from all macro definitions, and measured the sizes of the individual macro and language slices along with the combined slices. The given numbers are the average slice size values. We treat the set of toplevel macros specially so we count the toplevel macros into both the macro slice and the belonging vertices into the C/C++ slice as they belong to both kinds of slices.

	No s.	Macro s. size (avg)	C lang s. size (avg)	Combined s. size (avg)	Macro s. % (M/C lang)
flex	332	15.9	6369.2	6385.1	0.25
time	33	8.6	42.4	51	20.28
wdiff	25	7	270.3	277.4	2.59
ed	27	3.3	772.3	775.6	0.43

Table 2. Summary of forward slices

Backward slices may not necessarily contain macro calls. Although the number of macro calls is not so high in an average program, most of the backward slices contain macro calls. The percentage of the slices which contain macro calls in the analyzed projects are between 81.5% and 92%. The number of combined slices (which necessarily contain macros) and their average sizes are shown in Table 3, in which we used the same approach for measurement as with the forward slices. It can be observed that the backward macro slices are generally bigger than the forward slices, which can be explained by the fact that language slices usually contain much 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 may be that in backward case we produce slices for all vertices, so more of the large slices are counted, while in the forward case we selected only some 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 percents. The table shows 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 criteria the smaller the slice the better, of course not missing any dependency. Macro slices are safe in this sense, while still having relatively small added percentage.

It can be observed that in both directions the added code lines by macro slices are relatively small compared to the language slices, so one could argue about their usefulness. However, we believe that in many cases exactly these additions may be crucial from program comprehension point of view. In this respect, traditional C/C++ language slices can be treated as *unsafe*, missing important information. This can be well illustrated by

	No s.	C lang s. size (avg)	Macro s. size (avg)	Comb. s. size (avg)	Macro s. % (M/C lang)
flex	12376	9498.3	1774.7	11273.1	15.74
time	603	203.2	18.6	221.8	9.15
wdiff	1001	287.6	44	331.6	15.3
ed	3860	1884	37.4	1921.4	1.99

Table 3. Summary of backward slices

the following example taken from the *flex* subject program. The size of the combined forward slice of macro `reallocate_integer_array(flexdef.h:686)` is 8271. Macro slice takes only 67 from this size. 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

```
PUT_ON_STACK (ns);
```

but when the source is preprocessed, it is replaced by a do-while loop which is 358 characters long. The combined slice goes through 31 toplevel macros, from which the above mentioned is just one. This way the dependencies for the loop would be missed from the slice if macro slices would not have been used.

5. Related work

There are relatively few slicing tools available for C/C++ programs. Binkley and Harman [2] conducted an empirical study about static slice size of C programs and they mention three general purpose tools: Unravel [21], Sprite [15] and CodeSurfer [11], using the latter in their experiments. Unravel was a research prototype developed in a discontinued project. It includes a number of deficiencies including that it only accepts preprocessed ANSI C code, which makes it obvious that handling macros is not implemented. Sprite implements some enhancements to traditional slicing algorithms most notably in the field 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 with this approach.

The commercial slicing tool CodeSurfer, marketed by GrammaTech Inc., is probably the most current and mostly developed slicing program for C/C++ as of today. It is able to compute various static dependency data 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 usages and present these data to the user. However, it is not possible to compute slices from macro definitions as criteria, furthermore the slices will include only statements that exist after macro expansion. Nevertheless, we used this tool in our experiments since the information supplied by CodeSurfer about macro usage was sufficient to implement our approach.

The Ghinsu software maintenance environment is the most closely related tool to our approach [14]. With it, by clicking on a macro invocation the called definitions are highlighted (backward macro slice using our terms). Furthermore, it also supports both static and dynamic slicing, ripple analysis and other program analyses on ANSI compliant C source code. This tool also utilizes a dependency graph in which the tokens of preprocessed code are classified according to whether and how they are involved in macro expansion. Unfortunately, it seems that this project has been discontinued, and based on the latest information that we were able to find the implementation has some drawbacks which disables its use for real programs. For example, certain language features and complex projects consisting of multiple source files are not handled.

There are also some other tools which are not particularly slicers but involve similar functionality for the comprehension of macro usage. The GUPRO program understanding framework implements a macro folding mechanism, where a macro can be hidden or revealed at the place of the call [13]. The Understand for C++ reverse engineering tool provides cross references between the use and definition of software entities [20]. 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 in some situations. These tools however do not involve C/C++ language slicing as we propose in our approach.

6. Conclusion and future work

The work presented has been 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 the program comprehension point of view, existing methods are often incomplete. For example, 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 either provide rich features to model macro usages, but not both. This could have a negative impact on various fields related to program comprehension and maintenance in general. For instance, in change impact analysis, a failure to identify a dependency of a change could have the effect of inaccurately predict the cost of changes and of performing incomplete change propagation, which in turn results in increased risk of regression [17].

With this work we fill this gap and propose a *combined* approach to computing slices in C/C++ programs. We support the approach with a realistic sample program compre-

hension problem. Existing tools were employed in an experimental tool setup with which a number of program slices have been computed. We counted the program points returned by the combined approach and compared it to slices without the preprocessor components.

Our first results measured on open source projects are promising, undoubtedly showing the benefits of the approach. However, the method needs to be refined and larger case studies should be performed. We plan to qualitatively evaluate the approach in more depth to find out usage scenarios in which it would be most beneficial. We also plan to conduct more experiments with much bigger systems, like the Mozilla source code. But even in this phase of the research we definitely suggest to integrate similar combined strategies for slice calculation in existing tools like CodeSurfer. In it, it could be a possibility to use and extend the existing internal representation for this purpose. We plan to work in this direction as well in the near future.

Acknowledgements

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

References

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

[2] D. Binkley and M. Harman. A large-scale empirical study of forward and backward static slice size and context sensitivity. In *Proceedings of the International Conference on Software Maintenance (ICSM'03)*, pages 44–53. IEEE Computer Society, Sept. 2003.

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

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

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

[6] M. D. Ernst, G. J. Badros, and D. Notkin. An empirical analysis of C preprocessor use. In *IEEE Transactions on Software Engineering*, volume 28, Dec 2002.

[7] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus – Reverse Engineering Tool and Schema for C++. In *Proceedings of the 18th International Conference on Software Maintenance (ICSM 2002)*, pages 172–181. IEEE Computer Society, Oct. 2002.

[8] FrontEndART Software Ltd. `http://www.frontendart.com`.

[9] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, 1991.

[10] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, UIUC, Oct. 2005.

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

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

[13] B. Kullbach and V. 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, Los Alamitos, 2001. IEEE Computer Society.

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

[15] M. Mock, D. C. Atkinson, C. Chambers, and S. J. Eggers. Program slicing with dynamic points-to sets. *IEEE Transactions on Software Engineering*, 31(8):657–678, 2005.

[16] K. J. Ottenstein and L. 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.

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

[18] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of ISSTA'94*, pages 169–183, Aug. 1994.

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

[20] `http://www.scitools.com`, 2007.

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

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

[23] L. Vidács, A. Beszédes, and R. Ferenc. Macro impact analysis using macro slicing. In *Proceedings of the Second International Conference on Software and Data Technologies (ICSOFT'07)*, pages 230–235, July 2007.

[24] M. Vittek. Refactoring browser with preprocessor. In *Proceedings of CSMR 2003*, pages 101–110, Benevento, Italy, March 2003.

[25] M. Vittek, P. Borovanský, and P.-E. Moreau. A collection of C, C++ and Java code understanding and refactoring plugins. In *ICSM (Industrial and Tool Volume)*, pages 61–64, 2005.

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

[27] J. 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, Feb. 2000.