Developer Support for Understanding Preprocessor Macro Expansions

László Vidács[1], Richárd Dévai[2], Rudolf Ferenc[3], and Tibor Gyimóthy[3]

[1] University of Szeged & Hungarian Academy of Sciences, Hungary
`lac@inf.u-szeged.hu`
[2] FrontEndART Software Ltd., Hungary
`devai@frontendart.com`
[3] University of Szeged, Hungary
`[ferenc|gyimothy]@inf.u-szeged.hu`

Abstract. In the age of advanced integrated development environments there is a lack of support for understanding preprocessor macros. The preprocessor has proven to be a powerful tool for decades, but the developer is still guided poorly when the debugger stops at a source code line containing macros. The main problem is that the developer sees the original code, while the compiler uses the preprocessed code in the background. Investigating the usually nested macro calls can be a labor intensive tasks, which increases the overall effort spent on development and maintenance. There are several possibilities to help the developer in similar situations, but these are rarely employed since the preprocessor has its own, separate language to be analyzed. We implemented a Visual Studio plug-in (AddIn) that provides hand-on information on macros to increase the productivity of developers during debugging or program comprehension tasks. We enhanced the idea of macro folding, a technique to show/hide macro names and values within the source code editor; and defined a graphical view for macro expansions. In the background precise dynamic analysis of directives takes place, so the hint given for the developers considers all kind of preprocessor constructs like macros in conditionals and concatenating operators in the replacement text.

1 Introduction

An empirical study shows that preprocessor directives make up a relatively high 8.4% of source code lines on average [2]. Beyond the fact, that the preprocessor is the only way to define multiple configurations of a C/C++ program, today a large amount of legacy code relies to directives. Although the preprocessor is useful for forward engineering and development, it behaves as an obstacle in case of program understanding and reverse engineering tasks. 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 developer sees. In many cases the two codes are markedly different. These differences make program understanding harder for developers and analyzers, and they can cause problems with program understanding tools.

The first point where the software developer is facing problems with macros is when a runtime error occurs at a source code line which contains macros only. The usual debugger stops at the line in question, but there is no information on what is the real code that the compiler used. Besides constant-like macros, many times the macro name is replaced by whole C/C++ loops or complex expressions spreading across several lines – all hidden from the developer. Furthermore, several macros have multiple definition depending on conditional directives, which makes it hard to find the actual definition manually. These labor-intensive activities can increase the overall effort spent on development or maintenance tasks. Unfortunately, widespread integrated development environments today, like the Visual Studio for C++ [11], give fairly limited support for the developer. As the preprocessor language is independent from the C/C++ language, the analysis of directives requires a separate analyzer, extra risk and effort for tool developers. Although the benefits of such extension are clear, developers are still forced to do workarounds when investigate macro calls.

In this paper we present the following contributions for supporting developers‘ understanding of macros:

– Enhanced macro folding in textual source code view, which helps in situations like debugging.
– Visualization of macro expansions in graphical code view, which gives overall picture of a full macro expansion.
– Both views are integrated as an AddIn (plug-in) in recent versions of Visual Studio.

The implemented AddIn enables step-by-step macro extraction to reveal each internal step that leads to the final macro replacement text. Furthermore, the graphical tree-like view of the expansion gives an overview on what happens during the expansion.

The paper is organized as follows: Section 2 introduces the textual and graphical views of macros and mentions the internal representation of preprocessor directives. Next, the AddIn is introduced through examples and screenshots in Section 3. Related work is discussed in Section 4, while conclusions are drawn in Section 5.

2 Visualizing Macro Expansions

2.1 Textual view – macro folding

Folding is an interactive extension of the textual view of the source code. The idea of folding in the context of preprocessing is presented by Kullbach and Riediger [7]. The base idea of folding is to define a textual area in the source code, which is hideable by the user. A fold is associated with a label; when the area is hidden (folded), then the fold label is shown. The folding mechanism may be presented by special characters (▼, ▲, ► and ◄) in the textual view of the program code. These special characters denote folded and unfolded state of the area, where `►Label◄` shows the folded and `▼Content▲` shows the unfolded state.

The folding mechanism was successfully employed within the GUPRO program understanding environment. Conditional expressions, file inclusions and macro expansions were presented as folds in the GUPRO environment. While the underlying metamodel was rather simple (consisting of nine elements) it was flexible to cover the above mentioned preprocessor constructs. Although the idea is presented several years ago, the modern integrated development environments still lack of folding support for preprocessor constructs.

The folding technique can be most appropriately applied to preprocessor macros. Let us consider a simple example of a macro constant `Z`. The label of the folded view is the name of the macro, which corresponds exactly to the state of the source code before preprocessing (see Listing 1.1). On the other hand, the unfolded view shows the value of the macro constant, which view corresponds to the state of the source code after preprocessing, as shown in Listing 1.2.

```
#define Z 5
int var = ►Z◄;
```

Listing 1.1: Simple macro folded – initial source code

$\Longleftrightarrow$

```
#define Z 5
int var = ▼5▲;
```

Listing 1.2: Simple macro unfolded – preprocessed source code

Function-like macros can be tracked similarly, except that further macro expansions may take place even in arguments – besides the macro body. As a more complex example let us consider the code shown in Listing 1.3, where macro `A` has two parameters and two macro names are passed as actual arguments. There are 3 macro replacements belonging to the full macro expansion of `A`, so it requires 3 steps to fully unfold the macro call (see Listing 1.4).

```
#define Z 1
#define Y 3
#define A(x,y) x+y
int var = ▶A(▶Z◀,▶Y◀+3)◀
```

Listing 1.3: Function-like macro example

2.2 Graphical view

While the folding mechanism enables to follow macro expansions in a step-by-step manner, it is not so appropriate to give an overall view of the macro expansion as a whole. It is usual to have even 8-10 expansion steps before reaching the fully expanded state in text mode. We propose a graphical notation for presenting full macro expansions as shown in Figure 1. The root of the tree is the actual compilation unit, which contains a macro call. The call is denoted by the root macro name. Each macro name in the graph has two children: the replacement value on the left hand side, and the actual text of the call on the right hand side.

```
Initial: int var = ▶A(▶Z◀,▶Y◀+3)◀
Step 1:  int var = ▶A(▼1▲,▶Y◀+3)◀
Step 2:  int var = ▶A(▼1▲,▼3▲+3)◀
Step 3:  int var = ▼1+3+3▲
```

Listing 1.4: Function-like macro expanded

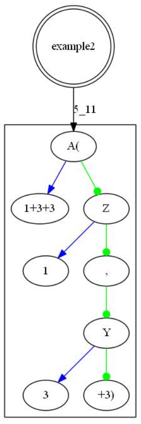


Fig. 1: Graphical hint for the function-like macro example

2.3 Internal representation

Macro expansions can be tracked at all places of programs, e.g. in conditional expressions as well. Detailed step-by-step macro expansion information is obtained through analysis of the preprocessor language. In our previous work we defined a schema (metamodel) for the preprocessor [18]. The Columbus Schema for C/C++ Preprocessing describes the original source code, the final preprocessed code and all transformation steps in between. Schema instances represent preprocessor constructs of concrete programs. We analyze one configuration at a time (dynamic instances.) Our representation contains all kinds of preprocessor construct, however in current work we use the macro-related part of the schema. Schema instances are produced by a tool, which can be smoothly incorporated into build processes, as it behaves as a usual preprocessor as well. The output of the tool is written out in XML format. Figure 2 presents the dynamic schema instance of the function-like macro example. Macro definitions are denoted with `(Func)Define` nodes. Definitions contain replacement text and may contain parameters. Macro calls are linked to active macro definitions via `(Func)DefineRef` reference nodes, which also mark actual arguments of function-like macros. Fold/unfold information can be extracted by traversing the instance graph along these references. For further information we refer to our previous work [18].

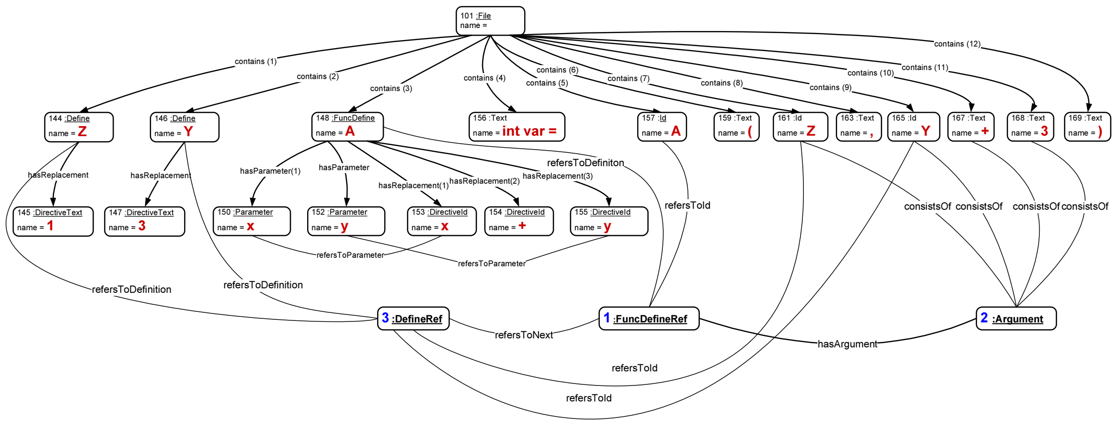


Fig. 2: Schema instance of the function-like macro example

3 IDE Plug-in for Understanding Macro Expansions

Surprisingly, modern IDEs lack of support for understanding directives. Visual Studio is a sofisticated development environment for .NET languages, with support for the C/C++ languages. There were enhancements made in recent years related to preprocessing, e.g. conditional directives are analyzed using a fuzzy parser and excluded conditionals are marked in the code editor, but there are still possibilities for improvements. Our goal was to extend the functionality of the Visual Studio by providing enhanced textual and graphical views based on the dynamic analysis of preprocessing directives. The base unit of the analysis is the compilation unit starting from the actual source file open in the editor. The whole compilation unit consists of the source file and all the header files included recursively.

We developed a plug-in for Visual Studio, which is called AddIn in its terminology. The architecture of the AddIn can be seen in Figure 3. The source code is first analyzed using the Columbus reverse engineering toolchain [4, 18]. The result is exported to XML (.ppml), which is processed by the AddIn: the node graph and the token trees are computed. At this point the AddIn reached its operating state and listens to user activities.

- Fold/Unfold action – the AddIn changes the source code view of the current file by replacing fold labels with their content and vice-versa.
- Expansion action – the AddIn generates `.dot` file form token trees and invokes the GraphViz tool to produce the layout of the picture. The generated image file, which is the output of the graphViz tool, is passed to the default image viewer tool of the system.

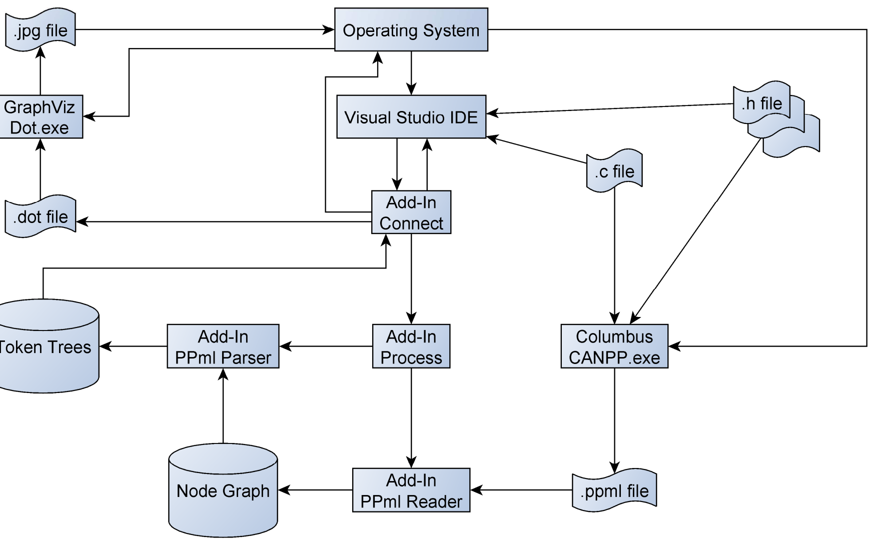


Fig. 3: Visual Studio AddIn architecture

The AddIn has a limited access to the internals of the Visual Studio. The AddIn has to work with the source code editor and has to read the properties of the actual project to access source files with full path and obtain command line defined macros for precise analysis. We have to note that some UI related features could not be implemented because a part of the API is for Microsoft internal use only. The AddIn has a simple but intuitive toolbar interface. The toolbar – with its two states and buttons – can be seen for Visual Studio versions 2005, 2010 and 2012 in Figure 4. The first button starts the `'A'`nalysis of the actual source file. As the analysis results are processed, the toolbar changes to operating state and annotates macros in the source code view. Folding mechanism and special notation of the code can be disabled any time using the `'HIDE'` button. Unfolding and folding buttons change the folding state of macros pointed by the actual cursor position. The `'EXT'` button decides between the two folding strategies. When a macro is unfolded, the replacement text of the macro is shown. In case of function-like macros, the actual arguments are also substituted. However, during the folding actions the original macro definition is not shown. In extended folding mode the definition of the macro is shown as an intermediate step, where the argument replacements can be followed before the final macro replacement step.

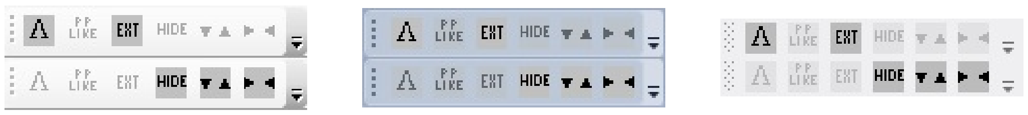
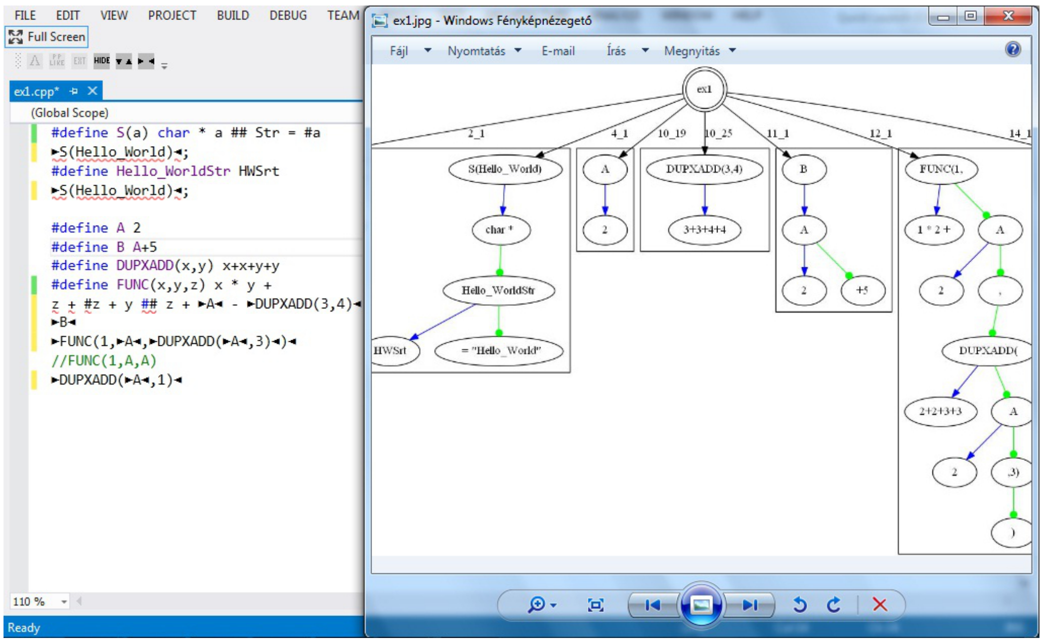


Fig. 4: VisualStudio AddIn Toolbar for versions 2005, 2010 and 2012 respectively

The inserted folding signs can be seen in the text editor of the development environment in Figure 5. Full macro expansions can be followed using a graph-based visualization as well. The notation of the graphical view is as follows. The shape of a node shows its node type: compilation unit (double circle), text (ellipse), conditional directives (hexagon). The edge colors denote the following properties: black edges show initial macro calls, where the macro position in the file is an attribute; green edges denote the chain of a full macro expansion, which gives the final result text of the call; blue edges denote macro evaluation steps; and red edges denote the conditional hierarchy if relevant.



```
#define S(a) char * a ## Str = #a
►S(Hello_World)◄;
#define Hello_WorldStr HWSrt
►S(Hello_World)◄;

#define A 2
#define B A+5
#define DUPXADD(x,y) x+x+y+y
#define FUNC(x,y,z) x * y +
z + #z + y ## z + ►A◄ - ►DUPXADD(3,4)◄
►B◄
►FUNC(1,►A◄,►DUPXADD(►A◄,3)◄)◄
//FUNC(1,A,A)
►DUPXADD(►A◄,1)◄
```

Fig. 5: Folding and macro expansion example in Visual Studio

First evaluation is done on test cases which cover most special macro constructs. The tool has proven to expand and fold/unfold macro constructs not only in C/C++ code but also in preprocessor conditional expressions and includes, which validates general usability. On the other hand, the tool is currently a research prototype, there are several points where the user experience of the tool could be greatly improved.

4 Related Work

Preprocessor directives are still widely used as no real size program with configurations exist without them. Ernst, Badros and Notkin [2] analyzed 26 commonly used Unix software packages and found that preprocessor directives made up the relatively high 8.4% of lines on average.

To overcome the preprocessor as a barrier in program understanding, researchers tackled problems of various areas. Analysis and visualization of include directives is a research topic from the early years, while in a recent work Spinellis [15] proposes a solution for the automatic removal of unnecessary includes, based on computed dependencies of program elements.

Dealing with software configurations is a well studied topic. Krone and Snelting [6] proposed concept lattices to aid reengineering configurations. Latendresse [8] proposed a symbolic evaluation algorithm for finding the conditions required for a particular source line to get through the conditional compilation. CViMe and C-CLR tools are Eclipse plugins, which collect and present configuration-controller macros [13]. Sutton and Maletic implemented analyzer tools on the top of the srcML infrastructure to reveal portability issues based on include files and configuration macros [16]. In the work of Garrido the analysis of preprocessor constructs was integrated into the C refactoring tool, where she implemented a configuration independent solution [5]. Livadas and Small developed a special preprocessor inserting special lines into the preprocessed file to support the source code highlighting methods of the Ghinsu program slicing tool [9].

Those working on C or C++ analyzers are confronted by the problem of preprocessor directives. Therefore, a lot of effort has been made to avoid their usage. Mennie and Clarke proposed a method to transform some macros and conditionals into C/C++ code [10]. Spinellis tackled the problem of global renaming of variables, preprocessor-aware solutions have been implemented in the CScout tool [14]. Saebjoernsen et al. [12] propose a mapping between the C language and the preprocessor to find inconsistent macro usage. The preprocessor-problem occurs also in the context of aspect mining and aspect-refactoring. Adams et al. worked on the problem of aspect refactoring, and also how to refactor various conditional compilation usage patterns into aspects [1]. In our previous work, we defined the macro dependency graph (MDG) for dependence based slicing of preprocessor macros [20]. Using the MDG C++ slices were extended with macro slices and better precision is achieved in case of more than 75% of backward slices [19]. Despite the wide range of initiations, current software development tools lack of support for the developers. In a recent paper Feigenspan et al. investigate the use of coloring techniques depending on the preprocessor conditionals in the FeatureCommander tool [3].

The initial work of Kullbach and Riediger on folding is already mentioned. The primary difference between the GUPRO environment and our solution is their purpose. While we concentrate on helping the developers with macros in place in the IDE, the GUPRO environment is a separate tool. Our internal structure (preprocessor schema) is capable of detailed analysis of preprocessor constructs, which is utilized also in macro expansion visualization to give an overview on what is happening. The Understand for C++ reverse engineering tool provides cross references between the use and definition of software entities [17]. This includes the step-by-step tracing of macro calls in both directions as well. The tool is appropriate for tracking back the uses of a give macro definition but the information is imprecise in certain situations like macro calls generated by `##` operators. A similar solution to macro folding is implemented in the Emacs editor. In C-mode, the `M-x c-macro-expand` command in Emacs will run the C preprocessor on the actual region and display the results in another buffer. This is similar to unfolding the actual macro. While this is a generic and simple solution, the folding mechanism is much more intuitive for stepwise investigation of a full macro expansion, than working with buffers. In addition, the visualization component is also available in our tool.

5 Conclusions

Developers need tool support when coping with preprocessor macros. Several research tools exist working with the preprocessor for various purposes, but the support for the developer is still limited in most common integrated development environments. We introduced an AddIn for Visual Studio to provide hand-on information on macro calls in the actual source code. We enhanced the idea of folding, a source code annotation technique, and integrated with graphical view of macro expansions. Macro information is obtained by detailed dynamic analysis of preprocessor constructs. The AddIn enables the step-by-step investigation of macro expansions and a quick view of the final form of the code, which is in fact compiled. Thus, developers need less manual effort for tasks related to macros – like debugging.

Future work includes enhancement of the user experience; improved, interactive macro graph that controls the source code view; and on field validation of the usability.

References

1. Adams, B., De Meuter, W., Tromp, H., Hassan, A.E.: Can we refactor conditional compilation into aspects? In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development. pp. 243–254. ACM, New York, NY, USA (2009)
2. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering 28(12) (Dec 2002)
3. Feigenspan, J., Kästner, C., Apel, S., Liebig, J., Schulze, M., Dachselt, R., Papendieck, M., Leich, T., Saake, G.: Do background colors improve program comprehension in the #ifdef hell? Empirical Software Engineering (2012), to appear; accepted 12 Apr 2012
4. Ferenc, R., Beszédes, A., Tarkiainen, M., Gyimóthy, T.: Columbus - reverse engineering tool and schema for c++. In: Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002). pp. 172–181. IEEE Computer Society (Oct 2002)
5. Garrido, A., Johnson, R.: Analyzing multiple configurations of a c program. In: Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005). pp. 379–388. IEEE Computer Society (2005)
6. Krone, M., Snelting, G.: On the inference of configuration structures from source code. In: Proceedings of ICSE 1994, 16th International Conference on Software Engineering. pp. 49–57. IEEE Computer Society (1994)
7. Kullbach, B., Riediger, V.: Folding: An Approach to Enable Program Understanding of Preprocessed Languages. In: Proceedings of the 8th Working Conference on Reverse Engineering (WCRE 2001). pp. 3–12. IEEE Computer Society (2001)
8. Latendresse, M.: Fast symbolic evaluation of C/C++ preprocessing using conditional values. In: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003). pp. 170–179. IEEE Computer Society (March 2003)
9. Livadas, P., Small, D.: Understanding code containing preprocessor constructs. In: Proceedings of IWPC 1994, Third IEEE Workshop on Program Comprehension. pp. 89–97 (Nov 1994)
10. Mennie, C.A., Clarke, C.L.A.: Giving meaning to macros. In: Proceedings of IWPC 2004. pp. 79–88. IEEE Computer Society (2004)
11. Microsoft Visual Studio. `http://www.microsoft.com/visualstudio/` (2012)
12. Saebjoernsen, A., Jiang, L., Quinlan, D.J., Su, Z.: Static validation of c preprocessor macros. In: AOSD '09: Proceedings of the 8th ACM international conference on Aspect-oriented software development. pp. 149–160. IEEE Computer Society (2009)
13. Singh, N., Gibbs, C., Coady, Y.: C-clr: a tool for navigating highly configurable system software. In: ACP4IS '07: Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software. p. 9. ACM, New York, NY, USA (2007)
14. Spinellis, D.: A refactoring browser for c. In: ECOOP'08 International Workshop on Advanced Software Development Tools and Techniques (WASDeTT) (2008)
15. Spinellis, D.: Optimizing header file include directives. Journal of Software Maintenance and Evolution: Research and Practice 22 (2010)
16. Sutton, A., Maletic, J.I.: How we manage portability and configuration with the c preprocessor. In: Proceedings of the 23rd International Conference on Software Maintenance (ICSM 2007). pp. 275–284 (2007)
17. Understand for C++ homepage. `http://www.scitools.com` (2009)
18. Vidács, L., Beszédes, A., Ferenc, R.: Columbus Schema for C/C++ Preprocessing. In: Proceedings of CSMR 2004 (8th European Conference on Software Maintenance and Reengineering). pp. 75–84. IEEE Computer Society (Mar 2004)
19. Vidács, L., Beszédes, A., Ferenc, R.: Macro impact analysis using macro slicing. In: Proceedings of ICSOFT 2007, The 2nd International Conference on Software and Data Technologies. pp. 230–235 (Jul 2007)
20. Vidács, L., Beszédes, Á., Gyimóthy, T.: Combining preprocessor slicing with C/C++ language slicing. Science of Computer Programming 74(7), 399–413 (May 2009)