Service Layer for IDE Integration of C/C++ Preprocessor Related Analysis

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

[1] Department of Software Engineering, University of Szeged, Hungary
Devai.Richard@stud.u-szeged.hu, [ferenc|gyimothy]@inf.u-szeged.hu
[2] MTA-SZTE Research Group on Artificial Intelligence, Hungary
lac@inf.u-szeged.hu

Abstract. Software development in C/C++ languages is tightly coupled with preprocessor directives. While the use of preprocessor constructs cannot be avoided, current IDE support for developers can still be improved. Early feedback from IDEs about misused macros or conditional compilation has positive effects on developer productivity and code quality as well. In this paper we introduce a service layer for the Visual Studio to make detailed preprocessor information accessible for any type of IDE extensions. The service layer is built upon our previous work on the analysis of directives. We wrap the analyzer tool and provide its functionality through an API. We present the public interface of the service and demonstrate the provided services through small plug-ins implemented using various extension mechanisms. These plug-ins work together to aid the daily work of developers in several ways. We provide (1) an editor extension through the Managed Extensibility Framework which provides macro highlighting within the source code editor; (2) detailed information about actual macro substitutions and an alternative code view to show the results of macro calls; (3) a managed package for discovering the intermediate steps of macro replacements through a macro explorer. The purpose of this work is twofold: we present an additional layer designed to aid the work of tool developers; second, we provide directly usable IDE components to express its potentials.

1 Introduction

Preprocessor directives – like macros and conditional compilation – constitute an integral part of the source code of C/C++ software, especially when applications are built for several target architectures, or in case of software product lines where several parallel configurations exist in the code [11]. An empirical study on open source applications shows that preprocessor directives make up a relatively high 8.4% of source code lines on average [3]. 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 influence the code quality in case of directive-intensive programs. Heavy use of directives is usually considered harmful [17] and it results in weak code quality and maintainability. A large amount of work addressed the elimination of directive use, with only partial results. 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 definitions depending on conditional directives, which fact makes it hard to find the actual definition manually. These labor-intensive activities can increase the overall effort spent on development or maintenance tasks.

Widespread integrated development environments today, like the Visual Studio for C++ [14], 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, out of the box solutions are usually not shipped with IDEs, and the developers are still forced to do workarounds to investigate macro calls. In our recent work [25] we introduced a Visual Studio Add-In that utilizes preprocessor related analysis and presents macro folding information together with a static view on macro calls. This paper builds on previous results and takes into account two observations: first, macro folding required to alter the source code, which is not acceptable by developers; and second, dynamic views are much more usable in concrete scenarios than static views introduced previously. In this work we intend to keep the workplace of the developer clean, while adding small pieces of information in a targeted and dynamic way. In line with the philosophy of flexible development environments and extensibility mechanisms, we provide a service layer for directive related information. This means that we use a wrapper for our preprocessor analyzer tools and provide an API for IDE extension developers to access macro calls, definitions, etc. Visual Studio extensions introduced in this paper also use this service layer to demonstrate its functionality.

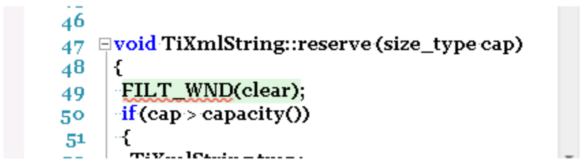
In this paper we present the following main contributions:

1. Service layer for preprocessor related analysis
2. IDE extensions built up on service layer
   - Macro and conditional directive highlights in the code editor
   - Side-by-side view to reveal the results of macro expansions
   - Macro definitions view to follow macro expansion internals

The paper is organized as follows: we briefly mention current IDE capabilities and propose additional views of the source code as a motivating example in the next section. Section 3 introduces extension mechanisms used in recent versions of Visual Studio, while in Section 4 we present our service layer to enable access for preprocessor related analysis results. We present new source code views as a demonstration of the usability of the service layer using various extensibility mechanisms in Section 5. Related work is discussed in Section 6, while conclusions and future plans are outlined in Section 7.

2 Motivation

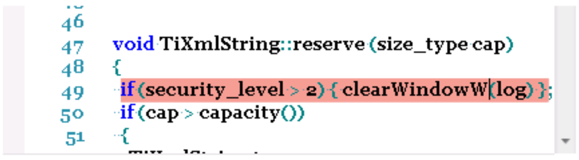
The source of program understanding problems is that the compiler gets the preprocessed code and not the original source code that the programmer sees. Let us consider a compiler error message pointing to line 49 in Figure 1:



```
46
47 void TiXmlString::reserve (size_type cap)
48 {
49   FILT_WND(clear);
50   if (cap > capacity())
51   {
```

Fig. 1. Example C++ code in Visual Studio code editor

To resolve the problem the developer has to look for the macro name in the code to find out the replaced text, which was actually compiled. The Visual Studio provides several ways for code search, one may search among files in the project, or even select to find classes or other program entities. Unfortunately the preprocessor language is a pure textual language and is unrelated to C or C++. Using the *Find references* feature one can find the `#define` directive of the `FILT_WND` macro. Yet, there can be several results, because the same macro name can be defined in several ways using conditional compilation. This enables to have different replacement texts for each platform or configuration. The next question is: Which one of the definitions is finally used? In recent versions the Visual Studio helps with conditional highlights, but even after finding the right definition, usually the search continues, because the macro definition contains further macro calls to look for. This is a tedious and time consuming task.



```
46
47 void TiXmlString::reserve (size_type cap)
48 {
49   if (security_level > 2) { clearWindowW(log) };
50   if (cap > capacity())
51   {
```

Fig. 2. Preprocessed code view using IDE extension

A better solution could be to alter the project configuration to produce the preprocessed file as well during the building process. One can compare the original and preprocessed code using `.i` files. However in this case the reason for the compiler error can be seen, but the concrete macro definition remains hidden, where the error could be corrected.

Our contributions include Visual Studio plug-ins to aid developers overcome these problems – like macro highlights in Figure 2. Furthermore, we provide a service layer to integrate preprocessor information into Visual Studio in a usable way for plug-in developers.

3 Visual Studio Extension Frameworks

In this section we first outline the history of plug-in possibilities of Visual Studio, and then briefly compare the most recent solutions in terms of their usability. The history of Visual Studio extension mechanisms goes back to the very first release. We distinguish four main types of extension mechanisms which were supported in recent releases:

- *Visual Basic for Application Macros*
- *Visual Studio Extensibility* and *Automation Add-Ins*
- *Managed Package Framework*
- *Editor Extension Point* components

The type of extension we should use is strongly depends on the objective we would like to achieve. From our point of view majority of interest lies packages, but our service layer can be consumed by any type of extensions. In this section the main historical changes of extensions are briefly introduced and at last we will give some direction about their common usage. We do not discuss *VBA Macros* as they were removed with the 2012 release.

To give a brief overview of changes in Visual Studio releases on extension frameworks we refer to Table 1. As shown in the table, development of *Add-Ins* stopped in the release in 2012, but generally prior versions brought only slight improvements of its API (we present Development Tools Environment (DTE) versions in the table). At the beginning *Add-In* APIs were poorly documented, but it was compensated by web sources – some driven also by Microsoft specialists – like *MZ-Tools*[1]. At the beginning, these resources approached from the Visual Basic side, while the C# support and the low number number of examples could be told more insufficient for beginners. There are also slight semantic differences between the usage and behavior of APIs for Add-Ins because of lingual differences, and one has to understand architectural flavors of the IDE to touch it the right place and with proper technique. After the opening of *Visual Studio Code Gallery* the support was improved by the increased number of examples. Search options and categorisation enhanced the proper use of different extension types. Open source projects both on *CodePlex*[2] and *GitHub*[3] meant a great help for developers in discovery of *Add-In* API's capabilities.

[1] MZ-Tools Add-In resources: http://www.mztools.com/resources_vsnet_addins.aspx
[2] CodePlex: https://www.codeplex.com
[3] GitHub: https://github.com

Table 1. Visual Studio extension type changes in recent releases

Release	2005	2008	2010	2012	2013
Add-In	DTE80	DTE90	DTE100		Deprecated
MPF	Release		WPF API		
EEP			Release		

For a deeper level of integration *MPF* can come into the picture. At first *MPF* was supposed to be used by Microsoft partners for commercial purposes. Only Microsoft's VSIP partners were able to distribute packages until the Visual Studio release in 2010. Hence its use was not spread as widely as in case of Add-In's. Users of MPF could rely only experienced partnership developer's blogs, but the Add-Ins remained first choice because of their simplicity. In 2010 Packages became freely available, but Add-Ins are still more supported by the community compared to them.

The newest and suppletory extension type for Visual Studio is *Editor Extension* as shown in Figure 1. Editor extensions are based on a lightweight plug-in technology called Managed Extensibility Framework which was also introduced in 2010. Their most important advantage is their simplicity. They can be used as editor enhancements to highlight code by formatting tokens through language classifiers, to add tags for background highlight, and to decorate code in the editor with custom UI elements using adornments.

In terms of API capabilities, packages enable access to deeper integration level, and besides they provide custom and consumable services through public APIs. Their main purpose is to extend Visual Studio with so called *Language Services*, hence they can be used to build up complex services as well. On the other hand, their use requires higher level of expertise. Packages are the base building blocks of the IDE and they even make it possible to reuse the IDE on other purposes with Shell Isolated packages like the *MS SQL Management Studio*. Partially the services provided for *MPF* components also available from *Add-Ins*, but they are designed to use a simple and lightweight API, so this is not a common method of their usage. The connection type of these two extensions are different at many points. It is advised by Microsoft from the release of 2013 to migrate *Add-Ins* to *MPF*. Except the simplified event system (also used for connection of Add-Ins) packages can use all API features provided for Add-Ins.

With the appearance of *Editor Extensions* one can access and easily extend the editor's presentation layer showing additional information to the programer in a lightweight method. Editor extensions owned pretty high interest as part of the reworked and *Windows Presentation Foundation* based IDE released in 2010. They give supplementary support for code highlighting and other feedback abilities just inside the editor window was depend on *Language Services* before, and can't be accessed through separated, lightweight API.

To summarize extensions by their most common usage, *Add-Ins* are the best choice whether we would like to extend the IDE with a standalone application as easy as possible. Add-Ins providing services directly to the user without a registered API, supported with controllers and tool windows to ease usability. An Add-In can use services of packages, and can manage commands registered either by itself or by other extensions. Even nowadays Add-In is a popular choice to create a simple extension component, however for complex applications it is more common to use MPF (as we also did). Editor extensions are mostly essential to give text decoration/highlight support, since their usage is limited.

4 Service Layer to Access Macro Expansions

The primary goal of our service layer is to extend macro-related capabilities of the Visual Studio IDE and to enable easier access to detailed preprocessor analysis. Our aim was to ease of its access from as many types of Visual Studio extensions as possible. While services registered by packages can be accessed form all type of extensions, we decided to prefer this technique over others. In the following we present an overview on the context and structure of the service, while its use is demonstrated in Section 5.

4.1 Service Layer Architecture and Context Outline

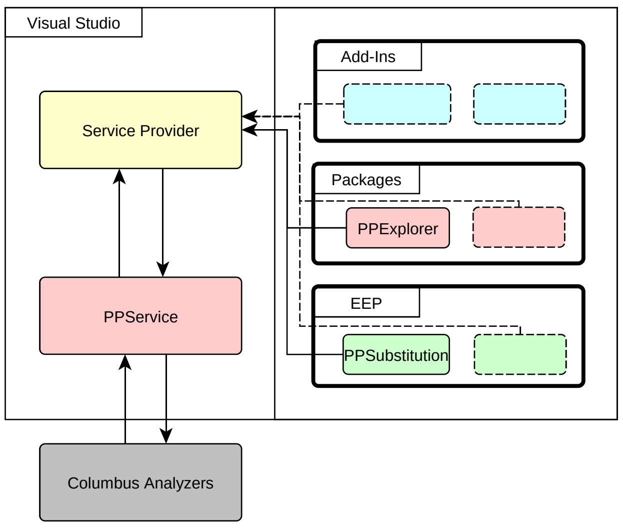


Fig. 3. Context and higher level architecture of the service layer

The Visual Studio extension structure is outlined in Figure 3. The central part of the figure is the Service Provider component of the IDE. It is responsible for dispatching services to various types of plug-ins. The *PPService* component represents our published layer of services and internal processes behind it. This component registers our service into the Visual Studio's service provider and also uses provided services. The core functionality of the *PPService* component is implemented by *Columbus Analyzers*. This component contains the *Preprocessor Analyzer* (CANPP) and builds our internal representation of the compilation unit (see 4.3 for more details). The service component depends on various data extractors like the Preprocessor ASG extractor as well.

The service is based upon different plug-in interfaces of Visual Studio that provided for MPF and Extensibility Framework. On the right hand side of the figure three actual types of plug-ins are represented. These plug-ins use our services through the provider component. This part of the figure presents a sample package and an editor extension we implemented to demonstrate the capabilities of the service. Note that our service is joining the service provider through an MPF interface, so the *PPService* component could also belong to the set of *Packages* on the right hand side of the picture.

4.2 Service Layer API

The service component hides the analysis process of directives and maintains a central repository for preprocessor details of the whole solution. The internal analysis can be triggered by some user action in the IDE (eg.: the build of a project). The component raises events about recent actions (eg.: about finished analysis), and API consumers are notified about these events. In case of course code changes new results invalidates previous data set of the actual project. The repository is designed for a later support of versioning mechanisms to track differences between analyzed versions. The current mechanism is connected to builds, which means that directive evaluation does not takes place during typing as the syntax highlights or intellisense, which use fuzzy parsers. Analysis results can be accessed either on demand based on actual file name and positions within the file, or the whole project structure can be traversed and the needed information can be collected by consumers.

In the rest of this section we give a brief description of our preprocessing data access interfaces presented in Figure 4. Interfaces are accessible through the *IPPAnalysisService*, which is registered by the package into the service provider. This interface is the base of data sharing, placed at the top of Figure 4. IContextProvider can be used to arrange analysis data to projects. Data can be extracted from *IDiscoveredProject* and an exact analysis round is presented by *IAnalysisConext*. The data for different source files are handled by *IAnalysisUnit*.

We would like to give the chance to a consumer to access simpler data without extracting details through complex traversals on our internal representation. Therefore results are accessible through different kinds of perspectives: a simpler and easy to use interface handling whole replacement texts, and a detailed interface where each token can be accessed.

Providing most basic type of data *ISubstitutionSpan* and its tracking version are available. We can access space information about all kind of macro names that are subjects of substitution process during macro processing. This interface

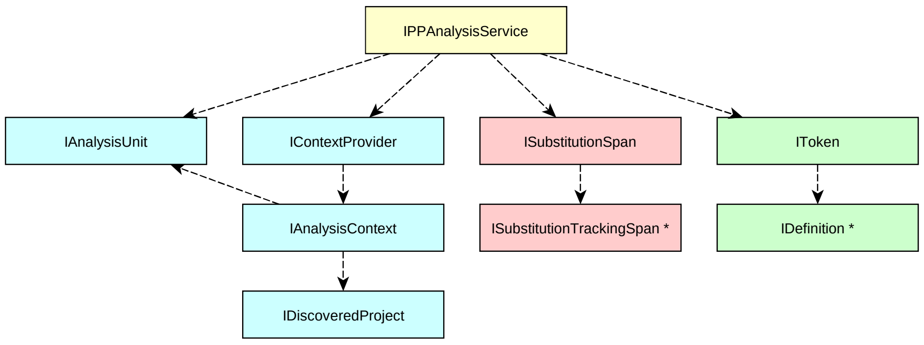


Fig. 4. Interface hierarchy of the service layer

gives information about macro calls and their substituted counter parts in code and preprocessed code file. Tracking version of the interface is ready to be used in Editor Extensions to create highlights on code files in IDE. These spans are also classified by different type of macro name areas like normal macro names and conditional subtypes as clauses evaluated to true or false, etc.

On next level the service provides data in tree structures. This interface group reflects our detailed internal representation (introduced in the next section below). These interfaces represent text tokens and their associated definitions used during macro substitution progress in a simplified structure. *IToken* interface gives information about the text, its file position, the substitution related child which is an other *IToken* calculated by the preprocessor during macro substitution. In addition, contains information about the used definition which lead to the child, finally information about whether the actual token is function or it is within the condition of a conditional directive. Definitions are presented through the *IDefinition*interface, which provides name and placement information. It also implements an interface called *IVsCodeDefViewContext* which makes the user able to feed the definition into the service of Visual Studio's Code Definition Window. Traverse of the token tree is also aided by several extension methods, although the actual analysis data is available easily through *IPPAnalysisService* by file name as well.

To present the usage of our interfaces we give a brief sample from the Macro Explorer managed package written in C# in Listing 1. A picture of this view in Visual Studio can be seen in Figure 8.

```
ISubstitutionSpan actSpan =
  _spans.FirstOrDefault(
    span => span.LineOriginal == line &&
      (span.ColumnOriginal <= column &&
       span.ColumnOriginal +
       span.LengthOriginal >= column));

if (actSpan != null) {
  IToken actToken =
    _tokens.FirstOrDefault(
      token => token.Line - 1 == actSpan.LineOriginal &&
      token.Column - 1 == actSpan.ColumnOriginal);

  if (actToken != null) {
    explorerTree.Items.Clear();
    TreeViewItem root = AddItem(null, actToken);
    if (actToken.UsedDefinition != null) {
      jumpCodeDefinitionToDefinion(
                          actToken.UsedDefinition);
    }
    explorerTree.Items.Add(root);
  }
}
```

Listing 1. Interface consuming code sample

In the example we search for a target span by line and column information. Whether we find the required span then we select the joining token from the actual token collection by the head column and line number of selected span. Than we clear the tree view shown by the Macro Explorer. Next we traverse the token tree and build the represented hierarchy with *TreeViewItems*. After that we give the definition of the root to the service of the Code Definition Window and at last we add the *TreeViewItem* created for the root to the tree view.

4.3 Internal Representation of Directives

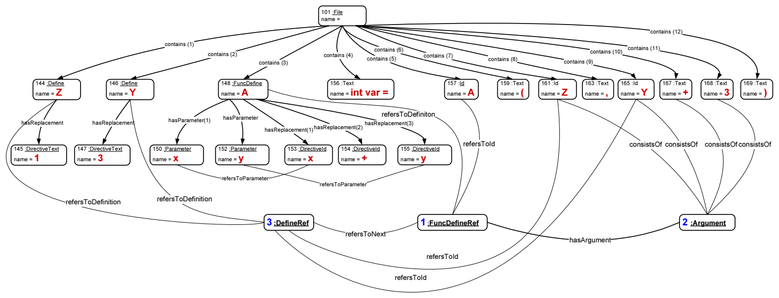


Fig. 5. Sample schema instance of a function-like macro

Detailed analysis is done by the analyzer tool of the Columbus Framework. In our previous work we defined a schema (metamodel) for the preprocessor [22]. 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. Using schema instances macro expansions can be tracked at all places of programs, e.g. in conditional expressions as well, in a step-by-step way. The output of the tool can be written out in XML format. Figure 5 presents the dynamic schema instance of a 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. Detailed macro call information, including all internal steps, can be extracted by traversing the instance graph along these references. These references take into account conditional compilation and concatenation and stringize operators used within macro replacement texts. For further information we refer to our previous work [22].

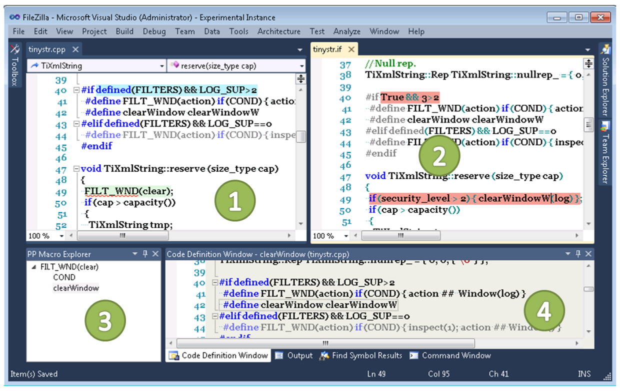
5 Applications of the Service Layer

In this section we present how the service layer is used to implement various mechanisms to support program understanding of preprocessor-related program parts. In our previous work we implemented a folding mechanism integrated into the code editor window [25]. The folded and unfolded states in the source code window correspond well to macro names (folded) and replacement texts (unfolded). Macro folds can also be nested in a way to show nested macro calls and even argument substitutions in case of function-like macros. The mechanism is appropriate for presenting the whole macro expansion process in a step by step manner. However due to practical reasons we needed to revise this concept. To implement folding in code edit window the actual source code must be modified in two ways: (1) folding markers (▼, ▲, ► and ◄) are inserted and (2) macros replacement text is replaced directly within the editor. To keep the code editor clean from unwanted modifications, a different approach is used in this paper. Code highlights are used to mark macros in the source code and a parallel window is shown to check the results of the macro replacement. In addition, the process of macro replacements can be followed using the macro explorer.

Figure 6 shows a Visual Studio screen, where all components of our tool set are present: (1) code editor window; (2) side-by-side view for parallel observation of the original code and the result of preprocesing; (3) macro definition explorer; and (4) macro definitions view.

5.1 Macro Highlights and Side-by-side View

As mentioned above, the motivation for highlighting code is to avoid changing the code just for presenting macro information. The aim is to give hands-on information around the code editor, but leave it untouched as much as possible.



```

#if defined(FILTERS) && LOG_SUP>2
 #define FILT_WND(action) if (COND) { actio
 #define clearWindow clearWindowW
#elif defined(FILTERS) && LOG_SUP==0
 #define FILT_WND(action) if (COND) { inspe
#endif

void TiXmlString::reserve (size_type cap)
{
 FILT_WND(clear);
 if (cap > capacity())
 {
  TiXmlString tmp;
```

```
//Null rep.
TiXmlString::Rep TiXmlString::nullrep_ = { 0,

#if True && 3>2
 #define FILT_WND(action) if (COND) { action
 #define clearWindow clearWindowW
#elif defined(FILTERS) && LOG_SUP==0
 #define FIL      D(action) if (COND) { inspec
#endif

void TiXmlStr  g::reserve (size_type cap)
{
 if (security_level > 2) { clearWindowW(log) };
 if (cap > capacity())
 {
```

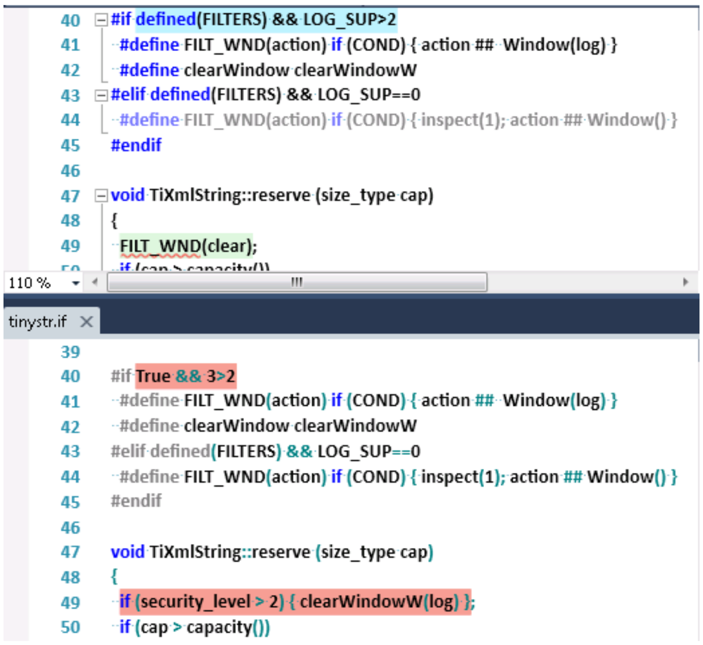
```

#if defined(FILTERS) && LOG_SUP>2
 #define FILT_WND(action) if (COND) { action ## Window(log) }
 #define clearWindow clearWindowW
#elif defined(FILTERS) && LOG_SUP==0
 #define FILT_WND(action) if (COND) { inspect(1); action ## Wind   }
```

Fig. 6. Macro related extensions built on the service layer in Visual Studio

The only change in the outlook of the code editor is that macro calls are highlighted (see the light green code in the left hand side of Figure 7). Code highlight positions can be obtained using the service layer API. The natural question that follows highlight is to check what is the final value of the macros after preprocessing. This is a central problem for a developer in a situation as outlined in our motivating example. The compiler in fact compiles the replaced text, which is not visible in a usual code editor. In case of an error message pointing at the position of the macro call, the developer needs to perform code search for the corresponding macro definition(s) to see what went wrong. Our service layer also can generate a copy of the source code for a user request where Macro names are being substituted. However in this format Macro calls are being replaced with the final macro replacement texts, but comments are held on to keep the code near to the origin version as possible. The *Macro Explorer* window is designed to be placed next to the code editor, thus parallel view of the original source code and the corresponding preprocessed code can be observed. Replaced code is highlighted using different color than in code editor as can be seen in Figure 7.

5.2 Macro Explorer and Definitions View

Macro highlights and the side by side view provide static information on macro replacements. Although observing the final preprocessed code is mostly sufficient, macro bodies may contain further macro calls, in many cases even 10-20 macros take part in a full macro replacement process. In these cases one may have to investigate internal steps of macro replacement. Searching manually for each macro definition is a time consuming task. Several macros have even more than one definitions surrounded by `#if` conditions. These conditions depend on macros as well, hence selecting the right one from several definitions is not straightforward. The macro explorer view and the code definition view is prepared for these situations. The macro explorer is a managed package and



```
#if defined(FILTERS) && LOG_SUP>2
  #define FILT_WND(action) if (COND) { action ## Window(log) }
  #define clearWindow clearWindowW
#elif defined(FILTERS) && LOG_SUP==0
  #define FILT_WND(action) if (COND) { inspect(1); action ## Window() }
#endif

void TiXmlString::reserve (size_type cap)
{
  FILT_WND(clear);
  if (cap > capacity())
```

```
#if True && 3>2
  #define FILT_WND(action) if (COND) { action ## Window(log) }
  #define clearWindow clearWindowW
#elif defined(FILTERS) && LOG_SUP==0
  #define FILT_WND(action) if (COND) { inspect(1); action ## Window() }
#endif

void TiXmlString::reserve (size_type cap)
{
  if (security_level > 2) { clearWindowW(log) };
  if (cap > capacity())
```

Fig. 7. Side-by-side view and macro highlight example in Visual Studio (two side windows shown below each other)

provides a hierarchical view of macro definitions in the order they take part in the macro replacement (see Figure 8).

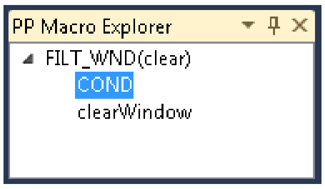
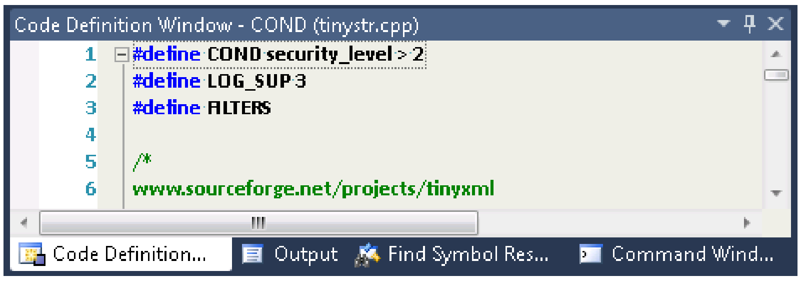


Fig. 8. Macro definitions explorer view in Visual Studio

Selecting any of the definitions in the explorer activates the code definition view to position to the selected macro definition in the source code (in the bottom part of the IDE). Note that definitions are usually placed in separate headers, not in the same file as the edited source code, where the macro is originally called. Several headers also take part in the include hierarchy, and the actual definition may be contained by them. The positioning also takes into account the conditional directives as can be seen in Figure 9.



```
#define COND security_level > 2
#define LOG_SUP 3
#define FILTERS

/*
www.sourceforge.net/projects/tinyxml
```

Fig. 9. Code definition window example in Visual Studio

Last but not least, the concatenating operator may cause strange constructs which are hard to find manually and also possibly result in coding problems and weak code quality. Using the concatenating operator (## in the replacement text of a macro, the macro parameter can be concatenated to a fixed string and the resulting token may produce a new, hidden macro call. These type of calls are rare, but could not be found by code search, as the macro name is not present in the code in its final form. The macro explorer helps developers overcome these situations as well, although our tool is currently a research prototype. Analysis of the Mozilla Firefox revealed 24 such concatenations which resulted in new macro calls, and these calls are used 337 times, which is not negligible.

6 Related Work

Preprocessor directives are still widely used as no real size program with configurations exist without them. Ernst, Badros and Notkin [3] 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 [19] 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 as well. Krone and Snelting [7] proposed concept lattices to aid reengineering configurations. Latendresse [9, 10] 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 [16]. Sutton and Maletic implemented analyzer tools on the top of the srcML infrastructure to reveal portability issues based on include files and configuration macros [20]. 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, 6]. The Refactoring Browser by Vittek [26] carries out automated modifications on a C source code. An interesting idea in this work is that of handling macros as special include files (the macro body is included), but handling of `##` operators is not solved in some cases. To handle the problem of configurations, this tool relies on user input. Livadas and Small developed a preprocessor inserting special lines into the preprocessed file to support the source code highlighting methods of the Ghinsu program slicing tool [12].

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 [13]. Spinellis tackled the problem of global renaming of variables, preprocessor-aware solutions have been implemented in the CScout tool [18]. Saebjoernsen et al. [15] 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 [24]. Using the MDG C++ slices were extended with macro slices and better precision is achieved in case of more than 75% of backward slices [23]. 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 [4]. 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 [8] and we applied the idea in our previous work. The folding mechanism was successfully employed within the GUPRO program understanding environment [2]. In this work we targeted the IDE and decided not to touch the code but use highlighting and parallel code window instead. The Understand for C++ reverse engineering tool provides cross references between the use and definition of software entities [21]. 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 a macro. Besides the folding mechanism shown in our previous paper, we provide parellel code window and a more intuitive view for stepwise investigation of macro definitions taking part in the expansion process.

7 Conclusions

Advanced integrated development environments influence the daily work of developers, thus having positive effect on productivity. IDEs also provide extension mechanisms to include plug-ins. Plug-ins enrich the environment to support several aspects of coding and help the developer maintain high code quality. In this paper our aim was to integrate preprocessor related analysis in Visual Studio. Built on our previous works on detailed macro analysis and IDE Add-Ins, we extend the capabilities of the Visual Studio with a service layer to provide access for other plug-ins to the internals of preprocessing. We presented the high level and low level architecture of this service and demonstrated its usability. We implemented packages and an editor extension to show the interoperability of mechanisms, and showed how these plug-ins can aid the daily work of developers. Using our services and tools one can overcome obstacles in program understanding caused by preprocessor conditionals, dependent macro definitions and concatenated macro parameters.

Our future plans include a usability study with the help of developers to identify directions of enhancements. We already identified rooms for development in integrating folding in code view, improving performance and implementing synchronized side-by-side view windows.

Acknowledgments

This research was supported by the Hungarian national grant GOP-1.1.1-11-2011-0006.

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. Ebert, J., Kullbach, B., Riediger, V., Winter, A.: GUPRO - Generic Understanding of Programs. In: Mens, T., Schrr, A., Taentzer, G. (eds.) Electronic Notes in Theoretical Computer Science. vol. 72. Elsevier (2002)
3. Ernst, M.D., Badros, G.J., Notkin, D.: An empirical analysis of C preprocessor use. IEEE Transactions on Software Engineering 28(12) (Dec 2002)
4. 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 18(4), 699–745 (2013)
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. Garrido, A., Johnson, R.: Embracing the c preprocessor during refactoring. Journal of Software: Evolution and Process 25(12), 1285–1304 (2013)
7. 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)
8. 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)

9. Latendresse, M.: Fast symbolic evaluation of C/C++ preprocessing using conditional values. In: Proceedings of the 7th European Conference on Software Maintenance and Reengineering (CSMR 2003). pp. 170–179. IEEE Computer Society (March 2003)
10. Latendresse, M.: Rewrite systems for symbolic evaluation of c-like preprocessing. In: Proceedings of the 8th European Conference on Software Maintenance and Reengineering (CSMR 2004). pp. 165–173. IEEE Computer Society (March 2004)
11. Liebig, J., Apel, S., Lengauer, C., Kästner, C., Schulze, M.: An analysis of the variability in forty preprocessor-based software product lines. In: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1. pp. 105–114. ICSE '10, ACM, New York, NY, USA (2010)
12. 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)
13. Mennie, C.A., Clarke, C.L.A.: Giving meaning to macros. In: Proceedings of IWPC 2004. pp. 79–88. IEEE Computer Society (2004)
14. Microsoft Visual Studio. `http://www.microsoft.com/visualstudio/` (2014)
15. 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)
16. 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)
17. Spencer, H., Collyer, G.: #ifdef considered harmful, or portability experience with C News. In: USENIX Summer Technical Conference. pp. 185–197 (June 1992)
18. Spinellis, D.: A refactoring browser for c. In: ECOOP'08 International Workshop on Advanced Software Development Tools and Techniques (WASDeTT) (2008)
19. Spinellis, D.: Optimizing header file include directives. Journal of Software Maintenance and Evolution: Research and Practice 22 (2010)
20. 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)
21. Understand for C++ homepage. `http://www.scitools.com` (2009)
22. 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)
23. 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)
24. Vidács, L., Beszédes, Á., Gyimóthy, T.: Combining Preprocessor Slicing with C/C++ Language Slicing. Science of Computer Programming 74(7), 399–413 (May 2009)
25. Vidács, L., Dévai, R., Ferenc, R., Gyimóthy, T.: Developer Support for Understanding Preprocessor Macro Expansions. In: Proceedings of International Conference on Advanced Software Engineering & Its Applications (ASEA 2012). pp. 121–130. Springer-Verlag (Nov 2012)
26. Vittek, M.: Refactoring browser with preprocessor. In: Proceedings of the Seventh European Conference on Software Maintenance and Reengineering (CSMR 2003). pp. 101–110. Benevento, Italy (March 2003)