Columbus Schema for C/C++ Preprocessing

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

Abstract

File inclusion, conditional compilation and macro processing has made the C/C++ preprocessor a powerful tool for programmers. However, program code with lots of directives often causes difficulties in program understanding and maintenance. The main source of the problem is the difference between the code that the programmer sees and the preprocessed code that the compiler gets. To aid program comprehension we designed a C/C++ preprocessor schema (supplementing the Columbus Schema for C++) and implemented a preprocessor which produces both preprocessed files and schema instances. The instances of the schema may be used to model: (1) preprocessor constructs in the original source code, (2) the preprocessed compilation unit, and (3) the transformations made by the preprocessor.

Keywords

C, C++, preprocessor, program understanding, schema, tool interoperability, Columbus

1 Introduction

Although the C/C++ preprocessor and the C/C++ compiler are separate, their usage is linked together from the start. The preprocessor has proven useful to programmers for over two decades, but it has also a number of drawbacks. The fundamental problem about preprocessing from a program comprehension point of view is that the compiler gets the preprocessed code and not the original source code that the programmer sees. In many cases the two codes are markedly different (according to [3] 8.4% of the source code they studied consists of preprocessor directives). These differences make program understanding harder for programmers and analyzers, and they can cause problems with program understanding tools. In the next section we will provide a concrete example.

Several researchers have been working in this area. A valuable contribution was made by Badros and Notkin [1], their framework allowing the user to write Perl callback functions to follow the work of the preprocessor (even in conditionally excluded code). The Ghinsu tool (coordinate mappings are used to describe macro calls [12]) and the GUPRO program understanding environment (a fold graph is constructed that contains information for visualizing directive usages [10]) are also remarkable solutions, but conditionally excluded code is not analyzed.

Supplementing the Columbus Schema for C++ [4] we designed a preprocessor schema to deal with the preprocessing in detail. To our knowledge it is the first publicly available general-purpose preprocessor schema. We hope that this work (like the Columbus Schema for C++) will be utilized as a reference schema for other works. The schema also describes conditionally excluded parts and may be used to aid overall program comprehension and understanding code in real cases as well. Possible applications include macro call-graph extraction, macro-expansion visualization, include hierarchy extraction and so on. The Columbus tool [5, 6] has its own preprocessor which, besides preprocessing, is able to generate instances of the schema. Largely thanks to this the mapping of the language elements to the original source code locations (e.g. where macro expansions are used) is improved in Columbus.

To facilitate tool interoperability the generated schema instances are also written in GXL format [7] so they can be used in software analysis, comprehension and maintenance tasks. Yet another application of the schema may be in code quality assurance. Code containing preprocessor constructs may be checked against constraints and rules (in general code with relatively simple macro complexity is better).

In the next section we will discuss the program code and preprocessor problem described above, which is introduced via an example, and also present our preprocessor schema. In Section 3 we give some example schema instances and ways they might be employed. A discussion of relevant articles and software tools are outlined in Section 4. Finally, in Section 5, we draw some conclusions and mention some ideas for possible future research.

Proceedings of the Eighth European Conference on Software Maintenance and Reengineering (CSMR'04)
1534-5351/04 $ 20.00 © 2004 IEEE

IEEE COMPUTER SOCIETY

2 The Columbus Schema for C/C++ Preprocessing

Preprocessing is the first stage of compilation, and is, in fact, totally separate from the compiler. Preprocessing means applying a set of low-level textual conversions on the source; the C and C++ language specifications ([8], [15]) have it in a separate section, and it is quite unrelated to the language syntax. These text-based transformations are hard to follow. This makes program understanding and maintenance difficult. For reverse engineers the preprocessor is similar to a black box. The connection between its input and output is well-defined, but in concrete, real-life cases it may be hard to see precisely what is going on.

The idea behind the preprocessing schema was motivated by the Columbus Schema for C++ [4]. The schema is an object-oriented model of preprocessor related language elements and their relationships. Object instances of the schema represent models of concrete source files, the resulting compilation units and the transformations made by the preprocessor. This will then establish a connection between the original code and the preprocessed code. During the preprocessing of a C/C++ source our preprocessor tool builds the schema instance of the compilation unit, which represents both the source code and the preprocessing transformations applied on it.

The preprocessed output of a given source code varies due to the interactions of conditional directives and predefined and command-line defined macros. We call these code-variations *configurations* (code belonging to one particular run of the preprocessor with a particular set of input macros). It is a far from trivial question of deciding how to handle these configurations in an analyzer tool, lots of other tools simply deal with the actual configuration.

To permit a wider range of information extraction we define two kinds of schema instances, with two ways of usage. The first is the *static instance* which does not depend on a given configuration (it will contain both true and false parts of an `#if` directive, etc.). The second is the *dynamic instance*, which is associated with one particular configuration, where conditional blocks ignored by the preprocessor are also omitted from the instance.

2.1 Motivating example

As an example for the preprocessor black box, let us consider the following code fragment of `math.h` taken from the Unix standard library.

```
#if defined __USE_MISC || defined __USE_ISOC99
...
#ifdef __STDC__
# define __MATH_PRECNAME(name,r) name##f##r
#else
# define __MATH_PRECNAME(name,r) name/**/f/**/r
#endif
#include <bits/mathcalls.h>
#undef __MATH_PRECNAME
```

One could start the investigation of this code as follows. The definition of the `__MATH_PRECNAME` macro depends on the `__STDC__` macro. `bits/mathcalls.h` is included and `__MATH_PRECNAME` is immediately undefined after that but, surprisingly, if we open the file `bits/mathcalls.h` in the source we can not find the text `__MATH_PRECNAME`. There are some questions raised by this code. Is the macro `__MATH_PRECNAME` used between the `#define` and `#undef` directives, or is this definition unnecessary here? Does the compiler really get this piece of code? If it does, which one of the two definitions is active? After some text searches in the standard inclusion directory, we find that `__MATH_PRECNAME` is present only in two headers. One of them is `math.h` where it is part of a definition of another macro: the code fragment below is in `math.h` but comes before the previous code.

```
#define __MATHDECL_1(type, function,suffix, args)
    extern type __MATH_PRECNAME(function,suffix)
        args __THROW
```

At this point we have to check whether or not this newly defined macro is present in `bits/mathcalls.h`, and we find that it is. But the following question still remains. There are two `#if` directives which come before the definitions of `__MATH_PRECNAME`. Is it possible that the compiler never gets this code? To answer this, we have to examine other macros to determine whether they are defined here, and what their values are. In general we can say that, to understand the code, the job of a preprocessor must be simulated by the programmer. Using our schema makes the whole procedure easier and a schema instance allows us to directly answer this and similar questions.

The outline of the dynamic schema instance of the example is shown in Figure 1 (only the relevant attributes are shown). As can be seen, `math.h` contains the definition of `__MATHDECL_1` (node 10 in the figure). This definition is used at least once in `mathcalls.h` (28), which can be checked by navigating through the *FuncDefineRef* object (40). `__MATHDECL_1` contains an invocation of the `__MATH_PRECNAME` macro (15), this invocation is connected (41) with its definition in `math.h` (22). It can also be seen in the figure that the first `#if` condition (18) is enabled (evaluated to true), and also the `#ifdef` of `__STDC__` (20) was true, so the first definition of `__MATH_PRECNAME` was active (22).

Now the questions listed at the beginning of this section can be answered in the following way. The `__MATH_PRECNAME` macro was used before it became undefined (so the definition is, of course, necessary), and the

IEEE COMPUTER SOCIETY

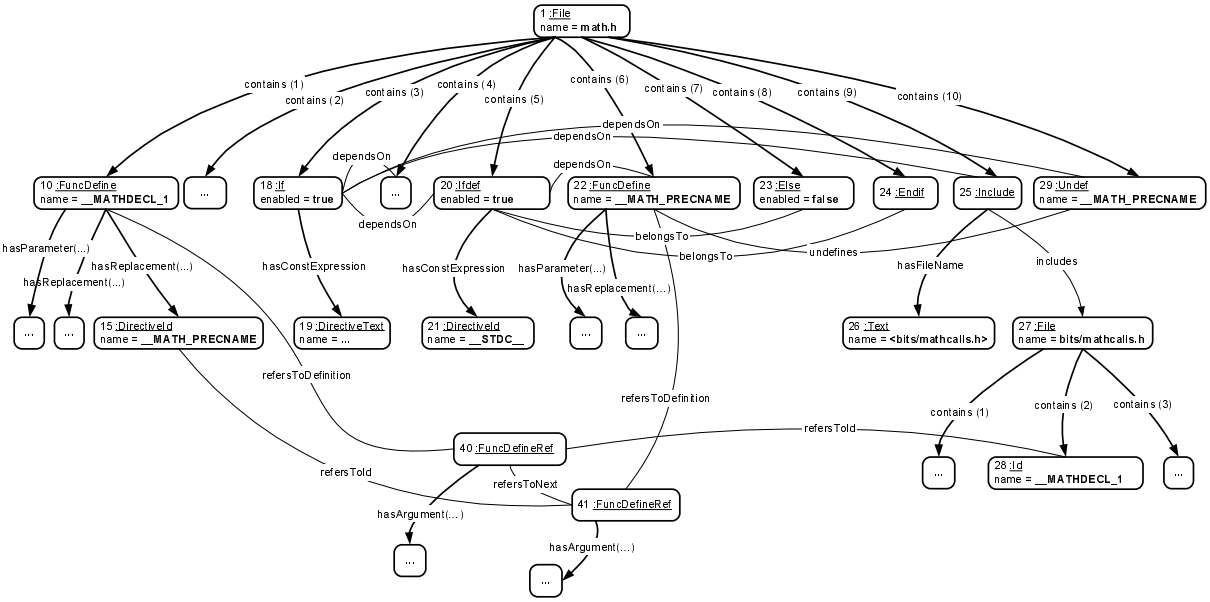

Figure 1. math.h: example dynamic schema instance

compiler gets this code fragment with the first definition. This is because the macro call (15) can be reached starting from *Include* node 25, which comes before *Undef* node 29.

As you might imagine from the example above there are a number of typical questions about the preprocessing task. For example: Where is the active definition of a macro? Is this file really included? Does the compiler get these lines after preprocessing? There are configuration independent problems as well, such as where all the different definitions of a macro can be found. Our aim with the present paper is to find a way of answering these questions and similar ones.

2.2 The structure of the schema

The schema is presented using the UML Class Diagram notation [13]. Both static and dynamic instances are described by the schema.

The UML Class Diagram of the preprocessor schema is given in Figure 2. The class *Base* is the abstract base class of all classes in the schema. Each element that appears in the source file has a position, so (except for *File*, *DefineRef* and *FuncDefineRef*) all classes are descendants of *Positioned*.

The root of an instance is a *File* object. A *File* object contains any number of ordered *Element* objects. *Element* is the abstract base class of elements contained in *File*. From the preprocessor's point of view a file consists of elements which can be either preprocessor directives or other text elements, so there are two specialized classes from *Element*: *Directive* and *Text*, the first also being an abstract one. Except for the text contained by directives, all textual elements in the file are represented by the class *Text*. The only parts of the source text of interest to the preprocessor are the identifiers (subclass *Id*), which are separate objects in an instance, because they may be macro calls. Otherwise the length and contents of text elements in one *Text* object is not determined by the schema, but by the strategy of instance building (it can be a preprocessing token or a longer sequence of characters).

Directives

Specialized classes of *Directive* correspond to the directive types. Most directives have various textual elements (like macro replacement) that are ordered lists of *DirectiveText* objects. The directives and their relations will now be described.

The *Include* directive includes a whole source file into the position of the directive. An *Include* object includes a new *File* object, which is the root object of all elements of the included source file (this part is also completely expanded, and it may also contain further included files). The *hasFileName* relation between *Include* and *DirectiveText* connects the filename with the include directive. There are two different types of aggregations between *Include* and *File* in the static and dynamic cases (see the constraint in Figure 2). In the dynamic case when a file is included several times in a compilation unit they require separate *File* objects with the whole subgraph, because there can be macro definitions between the different include directives (or in the included file) which can influence the included file bodies even in one configuration. In this case the relation is a composition. In the static case the file which

IEEE COMPUTER SOCIETY

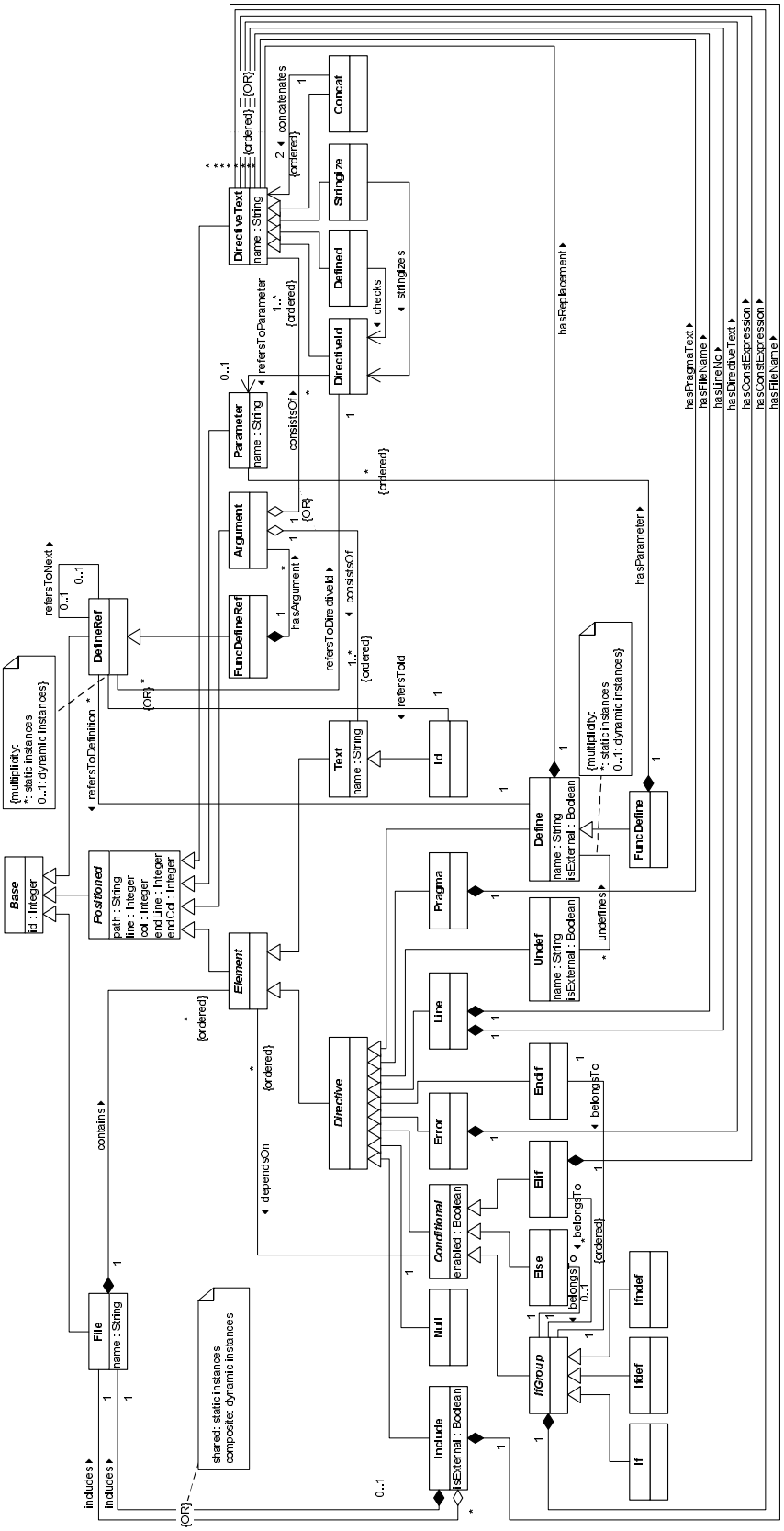

Figure 2. Class diagram of the schema

IEEE COMPUTER SOCIETY

is included several times has the same content because the static instance is configuration-independent so the one *File* object is shared among different *Include*s. To describe command line forced includes (this means that the file given in command line is included before the first line of source file) the class *Include* has an attribute called *isExternal*. For an example on the include directive see Figure 3 in Section 3.

Null directive represents a hashmark followed by a newline, it does nothing and its class has no relations.

Conditional directives represent code blocks controlled by the conditional code inclusion (commonly known as conditional compilation [8]). *Conditional* is the abstract base class of conditional directives which determine conditional blocks. Conditionals *If*, *Ifdef*, *Ifndef* are derived from the *IfGroup* abstract class. The conditional inclusion is controlled by special expressions called integral constant expressions [8]. These expressions must be evaluated in the preprocessing phase of the compilation. The result of the evaluation is an integer which is treated as a boolean value. They typically contain constants, macros and a special operator *Defined*. Operator *Defined* has one operand and evaluates to 1 if the operand is defined as a macro name, and to 0 if not. Only *IfGroup* and *Elif* can have constant expressions. The conditional block is a list of sequential elements beginning with an *If*, *Ifdef*, *Ifndef* or *Elif* and ending before the matching *Elif* or *Else* or *Endif* pair of previous directives (conditional directives can be nested). Each *Conditional* object has a conditional block, which is linked to the directive using the relation *dependsOn*, because these elements depend on it. (There may be additional conditionals or included files in a block.) Considering an *If-Elif-Else-Endif* sequence, the code of a conditional block is included into the preprocessed output file only if this block is the first in the sequence which has a conditional expression with value true. In this case the *enabled* attribute of the *Conditional* object is true, otherwise false (this attribute is relevant only in dynamic instances). To identify members of these conditional sequences the *belongsTo* relation is defined, so that *Elif*, *Else* and *Endif* objects can reference the appropriate *If* (or *Ifdef*, *Ifndef*) object.

Different configurations are due to conditional blocks, but a normal run of a preprocessor produces only a single configuration (this is modelled with a dynamic schema instance). For a software maintainer it is important to see more (all) configurations. Static schema instances enable all conditional blocks, and therefore at the same time information can be gathered from more configurations. For examples see Figures 4 and 5 in Sections 3.1 and 3.2, respectively.

An *Error* directive produces an error message and its usage is usually combined with conditional directives. It has *DirectiveText* elements after the directive name which are written out as error message of the preprocessor.

The *Line* directive has two tasks: it generates line information for the compiler and it redefines the `__LINE__` and the `__FILE__` standard C/C++ macros. *Line* has a line number and optionally a file name.

A *Pragma* directive is an implementation-defined control sequence for the preprocessor or the compiler (for example to disable warnings or prevent multiple header inclusions). It has directive-texts which may contain macro invocations.

The *Define* directive is used to define preprocessor macros. Classes *Define* and *FuncDefine* will be described in the next subsection.

The *Undef* directive makes a previously defined macro undefined. It references only the corresponding *Define* object. One definition can be undefined zero or more times (only the first is accepted, the *Undef* directives which try to undefine not defined macronames are simply ignored). The relation in the schema permits one *Undef* directive to reference multiple definitions, although in one configuration only zero or one definition can be referenced. Multiple relations are allowed only in the static case where all possible definitions (in different configurations) can be accessed (see constraint in Figure 2). *Undef* also has an attribute called *isExternal* for the command line undefinitions of built-in predefined macros.

Macros

Macro definitions are represented by classes *Define* for simple, and *FuncDefine* for function-like macros. *Define* has a *name* attribute, and replacement text which is to replace the macroname at the place of a macro call (this text is called a *replacement list*, and consists of *DirectiveText* objects). Definitions of function-like macros have zero or more ordered parameters. The formal parameter list is represented by objects of class *Parameter* (the opening and closing parentheses and the commas separating the formal parameters are not present in the schema).

All textual elements inside directives are represented by class *DirectiveText* (the text is stored in the *name* attribute). Identifiers have to be objects of class *DirectiveId*. Every macro replacement list may contain further macronames (*DirectiveId*) and may also contain *Concat* operators (##). The *Concat* operator concatenates the preceding and the following tokens into one new token.

The replacement list of function-like macros has some more specialities. In the list a *DirectiveId* object may refer to one *Parameter* object, this *DirectiveId* will be replaced with the corresponding argument during a macro call. In this kind of replacement list if the *Concat* operator concatenates a parameter then the parameter is substituted before concatenation, and further macro expansions can be made only after concatenation. The list may also contain *Stringize*

IEEE COMPUTER SOCIETY

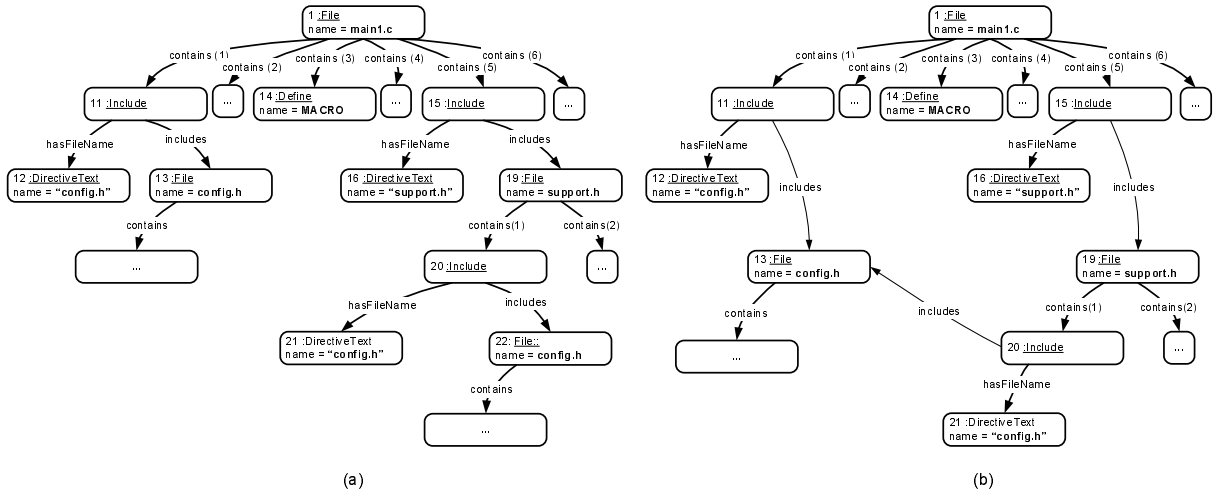

Figure 3. Dynamic (a) and static (b) schema instance of the example of include directive

operators (#). A *Stringize* operator must be followed by a *DirectiveId* which is a parameter name, and during the replacement the operator creates a string literal from the actual argument. The two replacement list operators are specialized from *DirectiveText* because during preprocessing both produce new text from the arguments.

Macro invocations are represented by *DefineRef* objects. A *DefineRef* object refers to an *Id* (in simple text) or *DirectiveId* (in the text of directives) and links it with its definition by referring to the appropriate *Define* object (objects of *DefineRef* do not represent any concrete source code element, they are helper objects). One macro definition can be used (referred to) zero or more times. Invocation of a function-like macro (*FuncDefineRef*) has arguments which are objects of class *Argument*. Arguments are texts that are separated by commas. According to the place of the call an argument consists of one or more *Text* or *DirectiveText* objects, or their *Id*s because the argument may contain further calls. Parameter substitution takes place after every macro in the argument list has been expanded but before other macros in the replacement list have been expanded.

The usage of *DefineRef* objects is different in the static and in the dynamic cases. In the static case a macro call (*Id* or *DirectiveId*) can refer to several definitions at the same time (relation *refersToId*), and this way all possible definitions can be tracked, which can be important for a maintainer. In the dynamic case a macro name (*Id*) can be linked only with its active definition (multiplicity is 0..1 in dynamic case, see constraint in Figure 2). At a given point in a source file the active definition of a macro is backward the first *Define* directive which has no matching *Undef* directive (the included source files are also taken into account). The macro names in the replacement list of a macro can contain further macros (*DirectiveId*). This identifier may be connected with more definitions even in the dynamic case. In the following example there are two expansions of macro `A` (lines 3 and 6). In the two cases different definitions of `B` will be active: in the first case the definition in line 1, and in the second case the definition in line 5.

```
1 #define B 3
2 #define A k*B
3 A                ==> k*3
4 #undef B
5 #define B 5
6 A                ==> k*5
```

This difficulty with nested macro invocations necessitated the introduction of the *DefineRef* class and its *refersToNext* relation. When the replacement list or any argument contains further macro calls the full expansion of a macro requires more *DefineRef* objects, which are linked to each other with the *refersToNext* relation. As it can be seen in the previous example, macro calls in a replacement list cannot be evaluated at the point of definition (macro call `B` in the replacement list of macro `A`). Once the macro expansion is started with an identifier, a list of *DefineRef* objects describes the first and the subsequent, generated macro calls. Each *DefineRef* object may refer to the next *DefineRef*, and each may be referred by zero or one object (the first has zero references). When a function-like macro is called, *DefineRef* objects for macro calls in arguments are included into the list before the further macro calls in the replacement list. The macro representation is further explained with an example in the following section.

3 Examples

In this section some examples are presented on how some commonly used preprocessor features can be mod-

IEEE COMPUTER SOCIETY

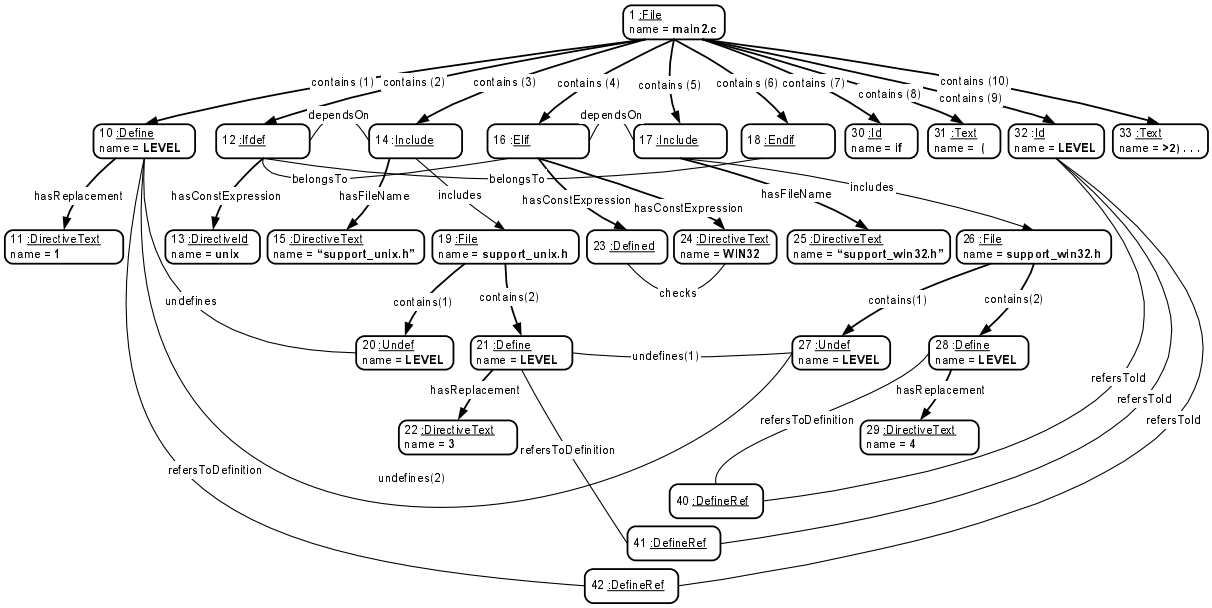

Figure 4. Static schema instance

elled using our schema. Details on static and dynamic instances are given.

Dynamic instances represent exactly one configuration, while static instances let us see the overall code without any specific information about a concrete configuration. We will illustrate this difference with an example of file inclusion. In static instances an included file name always produces the same *File* object, and each include directive refers to it. In dynamic instances every include has a different macro context and has its own *File* object, as mentioned in the description of *Include* directive in the previous section.

The dynamic and the static instance of the example code below can be seen in Figure 3.

```
  main1.c:
-------------------------
#include "config.h"
...
#define MACRO
...
#include "support.h"
...

  support.h:
-------------------------
#include "config.h"
```

The same file (`config.h`) is included twice, but in the second case it is via another included file. In the dynamic case the *File* object is directly contained (composition) by an *Include* object (11-13, 15-19, 20-22). In this example there are two *File* objects for `config.h` (13 and 22), this being caused by the different dynamic context of the two cases (e. g. macro definition 14 or the usual header protection construct *Ifndef–Define–*contents*–Endif*). On the other hand, static instances contain all conditional blocks regardless of the conditional expressions and contain all possible macro definitions and macro calls so the two *Include* directives of `config.h` (11, 20) share the same object (13).

3.1 Example for static instances

To learn more about static instances let us see the example below (see Figure 4 as well).

```
  support_unix.h:
-------------------------
#undef LEVEL
#define LEVEL 3
...

  support_win32.h:
-------------------------
#undef LEVEL
#define LEVEL 4
...

  main2.c:
-------------------------
#define LEVEL 1
#ifdef unix
#include "support_unix.h"
#elif defined WIN32
#include "support_win32"
#endif
if(LEVEL>2)...
```

The macro `LEVEL` is defined to be `1` by default (10), and there are two configurations: one for Unix and one for

IEEE COMPUTER SOCIETY

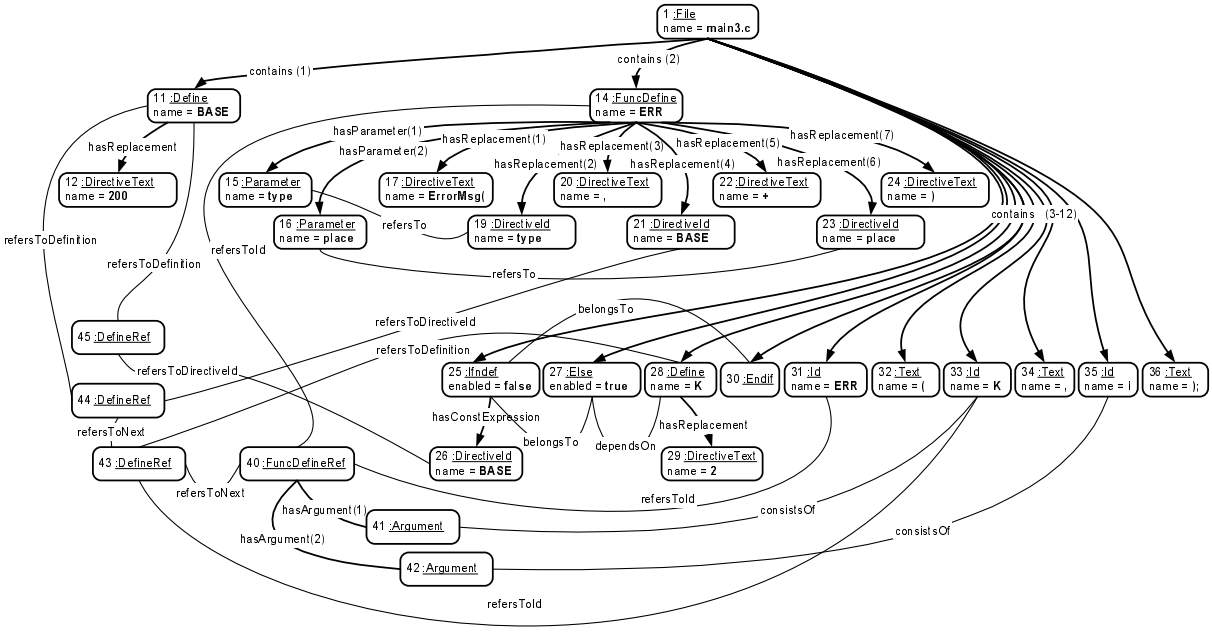

Figure 5. Dynamic schema instance

Windows. Both include supporting headers and redefine the macro `LEVEL` (21 and 28). After the directives the macro is called in a C if-statement. The macro call is connected through *DefineRef* nodes (40, 41, 42) with all the three possible definitions (10, 21, 28; using a pessimistic approach of the configurations). Definitions are also gathered from included files. In general, for a macro definition all possible macro calls can be seen and vice-versa, as well as, all possible definitions of a macro invocation (in all configurations). Similar to macro calls the *Undef* directives can refer to several definitions and one definition may be referred to by several *Undef* objects.

The details of static instances are not so well defined as those of dynamic ones. The strategy of building instances determines the final level of usability. In the previous example the *Undef* object 27 references two definitions (10 and 21). This is an example of a simple but safe building strategy. In fact, the reference to definition (21) is not possible. It will never be present in any configuration because *Undef* (27) and *Define* (21) are always in different configurations. An optimized building strategy should filter these types of relations out. In the future we will focus on implementing more intelligent strategies for building static instances.

3.2 Example for dynamic instances

A dynamic instance is an accurate description of a configuration. It is more precise than the static one: the *Undef* directive, say, points to exactly one position, and the actual macro calls can be followed.

The most interesting part of a dynamic instance is the macro expansion which makes use of a list of *DefineRef* objects. To see this, consider the following example code and its dynamic instance in Figure 5.

```
  main3.c:
-------------------------
#define BASE 200
#define ERR(type,place) ErrorMsg(type,BASE+place)
#ifndef BASE
#define K 1
#else
#define K 2
#endif

ERR(K,i);           ==>  ErrorMsg(2,200+i);
```

The macro `BASE` is defined to `200` (11). The function macro `ERR` (14) has two parameters (`type` and `place`; 15, 16), and its replacement text contains a function call using the two macro parameters. The *DirectiveId*s (19, 23) refer to the corresponding *Parameter* objects. In the code it can be seen that using conditional compilation the macro `K` is defined to 1 or 2 depending on wether the macro `BASE` is defined. The dynamic instance contains only the *enabled* conditional block (27), so the definition of `K` to 1 is not present in the instance. Apart from the enabled block, the disabled directives are also stored in the instance together with their constant expressions, but without their blocks. Storing these constant expressions is useful. For instance, in the example the *Else* part is enabled, but to see why, we have to look at the constant expression (with a macro; 26, 45, 11) belonging to the *Ifndef* directive, which is not en-

IEEE COMPUTER SOCIETY

abled. The relation *belongsTo* helps us find the matching conditional directives.

At the end of the example the macro `ERR` is called (31, 40) with `K` (33, 41) and `i` (35, 42) as arguments. The full macro expansion contains the objects 40, 43 and 44 in the list. *DefineRef* 43 links the first argument (macro call `K`) with its definition in the conditional block, so the actual argument will be 2 after substitution. The third *DefineRef* object (44) shows that, at the point of macro call `ERR`, the identifier `BASE` in the replacement list is a defined macro name. It is possible that later in the code the `BASE` may be redefined and the macro `ERR` is called. Then that call requires a new *DefineRef* object pointing to the new definition. Using the *DefineRef* and *Argument* objects the final result of the macro invocation can be easily got from a dynamic instance.

3.3 How to get information out of the instances

In general, information extraction requires graph walks in the generated schema instances. In the following we will present some typical applications.

Extraction of the preprocessed file from a dynamic instance requires the following actions during the walk. Text elements not depending on directives are simply written out to the output. The directives are not written out but instead all their effects are applied. This means that only the enabled conditional blocks are written out and the include directives are replaced with subgraphs. In addition, the macro substitutions are done by walking through each corresponding *DefineRef* object (at the same time argument substitution is performed and the required operators are applied).

Analyzing intermediate states of preprocessing helps us to better understand how the preprocessor works in a given situation. For instance the levels of macro expansions and whether the included subgraphs are placed instead of the include directives mean different intermediate states. To investigate nested macro calls and the levels of the substitution we can go through the list of *DefineRef* objects step-by-step (relation *refersToNext*), exchange the *Id*s with the appropriate replacement texts and substitute the parameters if needed. This technique is very similar to the folding approach in [10].

The include hierarchy of the compilation unit can be obtained from the static instance by simply traversing all the edges between the *File* and *Include* objects, starting from the *File* object of the input file.

As a last example we will show how the conditions under which a specific code line can get through the conditional compilation can be retrieved. Starting from the point of interest in the static instance, one should walk through the *dependsOn* and *belongsTo* relations up to the root *File* object. The result is the appropriate combination of the constant expressions of the traversed *Conditional* objects.

4 Related work

In the past decade preprocessing has been a frequent theme in the literature and this has led to the creation of several useful tools.

In spite of their disadvantages, preprocessor directives are still widely employed. Ernst, Badros and Notkin [3] analyzed the frequency and nature of preprocessor use. In their study they analyzed 26 commonly used Unix software packages written in C with about 970,000 source lines altogether (for example gcc, bash, emacs, gs, cvs, ...). Among other things they found that preprocessor directives make up the relatively high 8.4% of lines on average (varying from 4.5% to 22%).

A lot of related works deal only with some special aspects of preprocessing. Spencer and Collyer [14] investigated the use of conditional directives for separating codes running on different platforms. Their opinion is that the wide use of conditionals is "harmful" and should be avoided as much as possible. Well-organized code should be used instead. Krone and Snelting [9] analyzed the complex configuration structures created with directives and produced a graphical output of them. Latendresse [11] created a tool for finding the conditions needed for a particular source line to get through the conditional compilation. Vittek in [16] overviewed problems in refactoring in connection with the preprocessor. His Refactoring Browser makes automated modifications on a C source code. An interesting idea in this work is to deal with macros as special include-files (the macro body is "included"), but the handling of ## operators is unsolvable in some cases. To handle the problem of configurations, this tool relies on user input.

There are some studies that approach the preprocessing problem in a more general way. Badros and Notkin [1] constructed a framework which executes user defined Perl callback functions when an action of interest occurs during the preprocessing and parsing (after preprocessing they build an AST to deal with some C language-level constructs like call graphs). To do the preprocessing the authors modified and embedded the GNU C preprocessor library. As an example they wrote functions to describe macro expansions and also generated Emacs Lisp source to visualize them. With the help of hooks the conditionally excluded lines can be analyzed. However, one has to write custom code for each kind of use and this requires a good knowledge of some details of the tool's implementation.

In [10] Kullbach and Riediger worked along similar lines to us. They divided the code into foldable and non-foldable segments, which are visualized in the GUPRO [2] source code browser. Using this tool the important parts of the code can be seen much more clearly because the programmer can

IEEE COMPUTER SOCIETY

hide and show (fold/unfold) the segments. The fold/unfold structure is used to describe the preprocessor transformations (folded/unfolded state means the code before/after a preprocessor action). The structure is good for visualization and the user can also define custom folds. Since all transformations (macro calls, conditionals, etc.) are described using the same structure, this may be inconvenient for some other preprocessor related purposes. Yet another difference to our work is that GUPRO deals only with one configuration.

As part of the Ghinsu program slicing tool, Livadas and Small developed a special preprocessor [12]. They identified five mappings between the original and the preprocessed code. Their preprocessor inserts special lines into the preprocessed file to support Ghinsu's source code highlighting methods. Mappings for macro definitions and invocations are described in detail in their paper, but work on conditionally excluded code (i. e. configuration independence) had not yet been investigated.

In the present work we aimed to create the kind of schema that is able to include as much information as possible. The generated schema instances are not just for supporting our Columbus tool or for simply supplementing the Columbus Schema for C++; it is not limited to any special purpose. Because the preprocessor can create static and dynamic instances, the schema is useful for modelling the preprocessor constructs and their actual usage in a configuration.

5 Conclusions and further work

In this work we introduced the Columbus Schema for C/C++ Preprocessing. It supplements the Columbus Schema for C++, but can be employed separately as well. We have shown through various examples that different kinds of program analysis, comprehension and maintenance problems can be overcome by using instances of our schema. This is possible because we create dynamic and static instances by investigating the source code in a configuration dependent or independent way. The use of a standard notation and technology (UML, GXL) allows other reverse engineering tools to use the extracted information (for example source browsers, visualizers and code-understanding tools), so it relieves researchers of the burden of having to write preprocessors for different purposes and allows them to concentrate on their own concrete research topic.

For future research we plan to develop better methods for generating static schema instances which, say, provide a more accurate determination of possible define references of a macro usage. We have also started the work on designing a compact link between the Columbus Schema for C++ and the Columbus Schema for C/C++ Preprocessing.

References

[1] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analyses. *Software - Practice and Experience*, 30(8):907–924, 2000.

[2] J. Ebert, B. Kullbach, V. Riediger, and A. Winter. GUPRO - Generic Understanding of Programs. In T. Mens, A. Schrr, and G. Taentzer, editors, *Electronic Notes in Theoretical Computer Science*, volume 72. Elsevier, 2002.

[3] 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.

[4] R. Ferenc and Á. Beszédes. Data exchange with the Columbus Schema for C++. In *Proceedings of CSMR 2002*, pages 59–66, Budapest, Hungary, Mar. 2002.

[5] R. Ferenc, Á. Beszédes, M. Tarkiainen, and T. Gyimóthy. Columbus - reverse engineering tool and schema for C++. In *Proceedings of ICSM 2002*, pages 172–181, Montreal, Canada, Oct. 2002.

[6] Homepage of FrontEndART Ltd. `http://www.frontendart.com`.

[7] R. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. In *Proceedings of WCRE'00*, pages 162–171, Nov. 2000.

[8] International Standards Organization. *Programming languages — C++*, ISO/IEC 14882:1998(E), 1998.

[9] M. Krone and G. Snelting. On the inference of configuration structures from source code. In *Proceedings of the ICSM'94*, 1994.

[10] B. Kullbach and V. Riediger. Folding: An approach to enable program understanding of preprocessed languages. In *Proceedings of WCRE 2001*, pages 3–12, Los Alamitos, 2001.

[11] M. Latendresse. Fast symbolic evaluation of C/C++ preprocessing using conditional values. In *Proceedings of CSMR 2003*, pages 170–179, Benevento, Italy, March 2003.

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

[13] Object Management Group, Inc. *OMG Unified Modeling Language Specification*, version 1.5 edition, 2003.

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

[15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Third edition, 1997.

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

IEEE COMPUTER SOCIETY