Opening Up The C/C++ Preprocessor Black Box

László Vidács and Árpád Beszédes

Research Group on Artificial Intelligence, University of Szeged & HAS
Aradi Vértanuk tere 1., H-6720 Szeged, Hungary, +36 62 544145
lac@inf.u-szeged.hu; beszedes@cc.u-szeged.hu

Abstract. File inclusion, conditional compilation and macro processing has made the preprocessor a powerful tool for programmers. Preprocessor directives are extensively used in C/C++ programs and have various purposes. However, program code with lots of directives often causes problems 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 is given. We designed a Preprocessor Schema and implemented a preprocessor which produces both preprocessed (`.i`) files and schema instances. The Schema is general purpose and its instances model both the whole preprocessed compilation unit and the transformations made by the preprocessor. Therefore it facilitates program comprehension and tool interoperability.

Keywords

C, C++, preprocessor, program understanding, program model, Preprocessor Schema, tool interoperability

1 Introduction

Although the C preprocessor and the 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 quite different. These differences make program understanding harder for programmers and analysers, and they can cause problems with program understanding tools. In the next section we provide a concrete example.

Several researchers have been tackling this problem. The GHINSU tool (coordinate mappings are used to describe macro calls [10]) and the GUPRO program understanding environment (a fold graph is constructed that contains information about directives [8]) are remarkable solutions, but their usage is limited to

certain domains. We designed a Preprocessor Schema to deal with the preprocessing in detail, and made it available for other researchers to use. This is the first publicly available general purpose preprocessor schema. We hope that this extension to the Columbus Schema [3] will be utilised as a reference schema for other works. Furthermore, we also implemented a preprocessor tool (CANPP) which is able to generate instances of our schema. We also give examples on the way the schema can be utilised, and concrete program understanding scenarios.

In the next section we will discuss the program code-preprocessor problem mentioned above, which is introduced via an example. In Section 3 we furnish our solution, the Preprocessor Schema and example schema instances and usage scenarios. Related papers and software tools are outlined in Section 4. Finally, in Section 5 we discuss our conclusions and suggestions for future work.

2 Motivating example

Preprocessing is the first stage of compilation, and is in fact, totally separate from the compiler. Preprocessing means a set of low-level textual conversions on the source; the C and C++ language specification ([6], [16]) contains it in a separate section, but it has no connection with the language syntax. These text-based, unstructured transformations are hard to follow. This in turn makes program understanding and maintenance more difficult. Due to these transformations, 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.

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

```
#if defined __USE_MISC || defined __USE_ISOC99
...
# define _Mdouble_      _Mfloat_
# 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 _Mdouble_
# undef __MATH_PRECNAME
...
```

The definition of the `__MATH_PRECNAME` macro depends on the `__STDC__` macro. Then the file `bits/mathcalls.h` is included and after that this macro is undefined immediately. Surprisingly, if we open the file `bits/mathcalls.h` we cannot find this macro! There are some questions raised by this code. Is the macro `__MATH_PRECNAME` called between the define and undef directives? Is it a bug 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 and here it is part of a definition of another macro:

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

From this we have to check whether or not this newly defined macro is present in bits/mathcalls.h. Actually, it is. But we have not finished yet, because a question still remains. There are two #if directives 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 the job of a preprocessor must be simulated by the programmer to understand the code. Using our Schema makes the whole procedure easier and a schema instance enables us to directly answer these questions.

The outline of the schema instance of the example is shown in Fig. 1.

As can be seen, math.h contains the definition of __MATHDECL_1. This definition is used at least once in mathcalls.h as can be seen by navigating through the ***FuncMacroRef*** object B. __MATHDECL_1 also contains invocation of __MATH_PRECNAME macro, this invocation also being connected with its definition in math.h. It is also in the figure, that the first #if condition (C1) has the logical value true (1), and also the #ifdef of __STDC__ (C2) was true, so the first definition of __MATH_PRECNAME was active. After this, we can answer the set of questions at the beginning of this section in the following way. The __MATH_PRECNAME macro was used before it was undefined (so it is, of course, not a bug), and the compiler gets this code fragment with the first definition active.

As can be seen from the example above there are several typical questions about preprocessing. Where is the active definition of a macro? Is this file really included? Does the compiler get these lines after preprocessing? And so on. Our aim in the paper is to find a way of answering these questions and similar ones.

3 The Preprocessor Schema

During the preprocessing of a C/C++ source CANPP builds the Schema instance of the compilation unit, which represents both the whole source code and the preprocessing transformations made by CANPP. There are two ways of usage. The first makes use of operations/investigations on the original source code after preprocessing, the second follows the semantics of the whole preprocessing activity.

The preprocessed output of a given source code varies due to the predefined and command-line defined macros. We generate such models that belong to one particular run of the preprocessor (using a particular set of input macros). During the analysis we ignore the conditional blocks which have a false value. Just preprocessing these blocks without the corresponding set of macros would only give partial results.

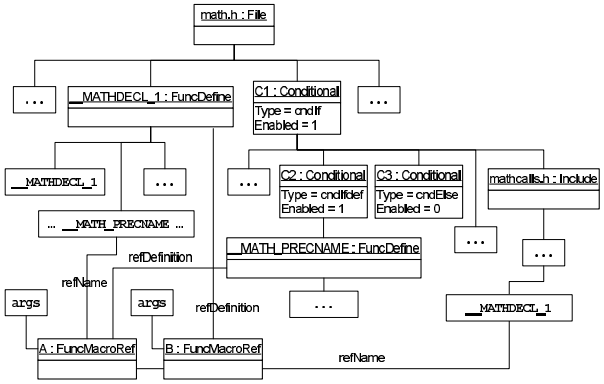

Fig. 1. math.h: example Schema instance - UML Object diagram

3.1 Structure of the Schema

The Schema instance is a special graph (a decorated tree), the structure of this graph being described by the the Preprocessor Schema. The schema is presented using the UML [12] Class Diagram notation. The class structure of the Schema follows the general structure of the C/C++ sources.

From our point of view a source file consists of elements which can be either preprocessor directives or other text elements (***Directive*** or ***Text*** classes in Figure 2). Text elements are divided into two groups: normal text elements and decorated text elements. The latter elements refer to macro definitions, parameters, etc. The UML Class diagram of the Preprocessor Schema is given in Figures 2 and 3. The root of the generated tree is one ***File*** object. A ***File*** object includes any number of ***Element*** objects. ***Element*** is an abstract base class of all elements in the file. There are two specialized classes from ***Element***: ***Directive*** and ***Text***, the first is also being abstract.

Preprocessor directive part. Specialized classes of ***Directive*** correspond to the directive types. These are the following:

- ***Empty*** - The Empty directive, does nothing, has no arguments and no following tokens. (code: `# newline`)
- ***Error*** - Produces an error message with the text tokens that follow it.
- ***Pragma*** - An implementation-defined control sequence. It contains text tokens, and possibly macro invocations.
- ***Include*** - An ***Include*** object contains a new ***File*** object, which is the root object of a subtree corresponding to the included source file (– this subtree is also completely expanded). It may also contain more included files. The include directive in the source is followed by a list of tokens. These tokens may contain macro calls, but after expansion the whole list must have a valid filename. This list of tokens is also contained in the ***File*** object.
- ***Conditional*** - Represents code blocks determined by the conditional inclusion (commonly known as conditional compilation). It has a ***cond_type*** attribute according to the conditional preprocessor directives (`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else` - note that `#endif` is not present in the Schema as a class). The first four directives are followed by an integral constant expression (see [6]). Constant expressions are evaluated during the preprocessing phase of compilation. The result of the given constant expression is stored in the ***Conditional*** object. 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 contains a conditional block. If the conditional expression has the logical value `true` (means not 0) after evaluation, the ***Conditional*** object becomes the root of elements in the block. There may be additional conditionals or included files in this subtree. Before or after an enabled conditional block there are some disabled blocks. Their elements are currently not included in the generated Schema instance (see example in Section 3.2).

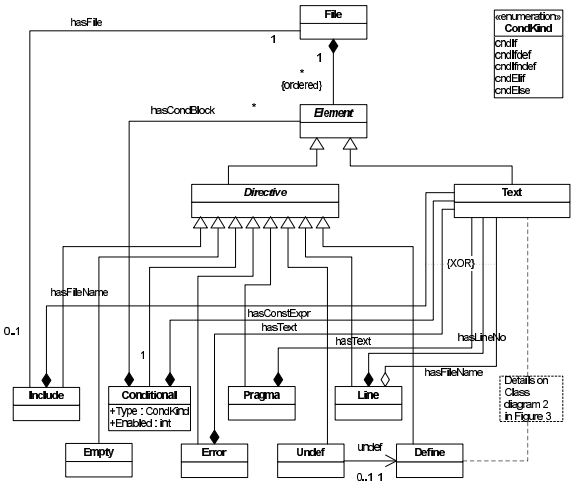

Fig. 2. Schema UML Class diagram - Directive part

- ***Define*** - This class is employed for a macro definition and it contains a replacement list of tokens. With function-like macros it also contains a parameter list (for detailed description see below).
- ***Undef*** - References only the corresponding ***Define*** object.
- ***Line*** - It is followed by a number and optionally a file name (these can be also described with macros). It has two tasks: it generates line information for the compiler and it defines the `__LINE__` macro (and the `__FILE__` macro if its text is given). Both are standard C/C++ macros.

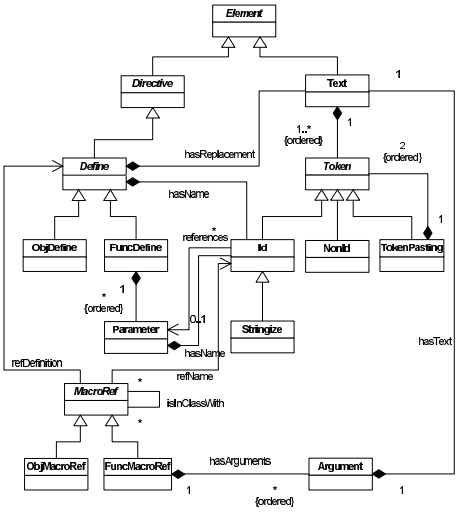

Fig. 3. Schema UML Class diagram - Text and Define part

The Text and Define part. ***Text*** is generally a list of ***Tokens***, while ***Token*** is an abstract base class of tokens. ***Id*** objects stand for identifier tokens (macro

names, macro parameters and simple identifiers). Other tokens are represented by ***NonId*** objects. There can be two special operators in a replacement list of a macro: stringize operator (`#`) and concatenate operator (`##`). Both produce new tokens from their arguments. The first creates a string literal from a macro parameter, while the second concatenates those tokens preceding and following it into one token. Class ***TokenPasting*** (`##` operator) is specialized directly from ***Token***, class ***Stringize*** (`#` operator) is specialized from class ***Id***, because its argument is a parameter of a function-like macro (see examples in the next section).

The ***Define*** class represents directives for a macro definition. It can define object-like and function-like macros (classes ***ObjDefine*** and ***FuncDefine***). A ***Define*** object has an identifier as its name, and has a replacement list (***Text*** object). This list can also contain ***TokenPasting*** operators. A ***FuncDefine*** object contains an ordered list of ***Parameter*** objects as well. In the latter case the replacement list can contain ***Id***s referencing to a Parameter, and can also ***Stringize*** operators followed by an ***Id***.

Macro invocations are represented by ***MacroRef*** objects. A ***MacroRef*** object links an ***Id***, which is a macro name at the point of occurrence, with its active definition. At a given point in a source file the active definition of a macro is the first `#define` directive before this point, assuming there is no `undef` directive between them (note: the included source files are also taken into account). A macro definition can contain additional macro invocations, these invocations cannot be evaluated at the point of the definition. The `isInClassWith` association gives a classification on ***MacrolRef*** objects. Once the invocation is started with an identifier, a set of ***MacroRef*** objects describe the first and the further, generated invocations; this set forms one class in the classification mentioned above. With the help of this reference set the texts can be replaced with their active definition even in case of nested macro calls. The invocation of a function-like macro (***FuncMacroRef*** object) in addition contains an ordered list of arguments. An ***Argument*** object is a ***Text*** which will replace the macro parameter in the replacement list.

3.2 Example Schema instances

In this section we present examples of how some commonly used preprocessor features can be modelled using our Schema.

Include directive. The following example code shows how `include` directives are modelled.

```
#include <stdio.h>
#include "config.h"
#include _SUPPORT_H
int main () {  ...  }
```

Include directives are represented by an ***Include*** object which contains a ***File*** object as the root of the subtree. So the generated tree can have recursive subtrees. The Schema instance of the above example is shown in Figure 4. This

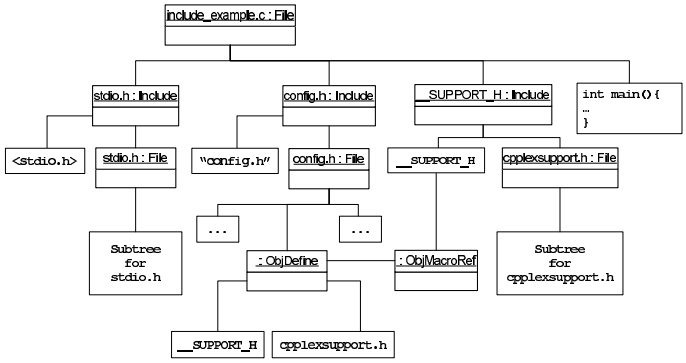

Fig. 4. Schema instance of the example of include directive

simple example contains an unusual feature, namely the macro call in the third include directive. This macro is defined in `config.h` and is expanded in the text `cpplexsupport.h`. The ***File*** object just corresponds to this header file (the usage of the ***ObjMacroRef*** object is described later).

Conditional code inclusion. The following example is about conditional compilation (here we assume that a value of 1.3 is assigned to the macro `__VERSION` and a value of 1 is assigned to the macro `__unix`).

```
#if __VERSION > 2
code1
#else
code2                          ==>         code2
#if defined __unix
code3                          ==>         code3
#elif defined _WIN32
code4
#else
code5
#endif
code6                          ==>         code6
#endif
```

The Schema instance belonging to this example is shown in Figure 5. The shaded parts have a `true` value after evaluation and they appear in the output as well. There is a straightforward connection between the program code and the diagram.

Macros. Our last example shows how various macro usages can be described: object-like macros, function-like macros, macro in replacement list, `#` and `##` operators. The example and its expansion can be seen below.

```
#define N 200
...
#define A(a,b) X ## b #b a+N
#if N <100
#define K 2
#else
#define K 1
A(K,3)                         ==>         X3 "3" 1+200
```

The schema instance of this code fragment can be seen in Figure 6. The definition of macro `A` consists of three parts, namely the name token, the parameters and the replacement tokens. The replacement tokens contain a concatenate operator with the first parameter and an ***Id***, a stringize operator with the second parameter, and the sum of the two parameters (***Id***, ***NonId***, ***Id*** tokens). The parameter occurrences are linked with the parameter definitions. These are the

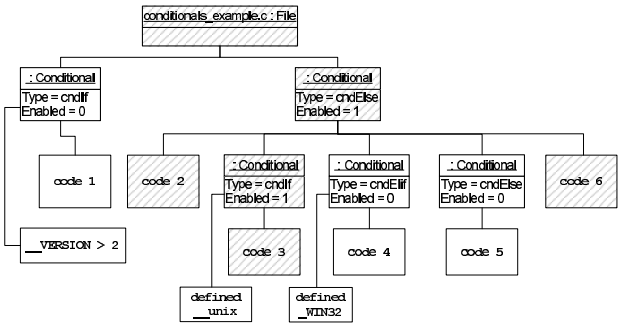

Fig. 5. Schema instance of the example of conditional compilation

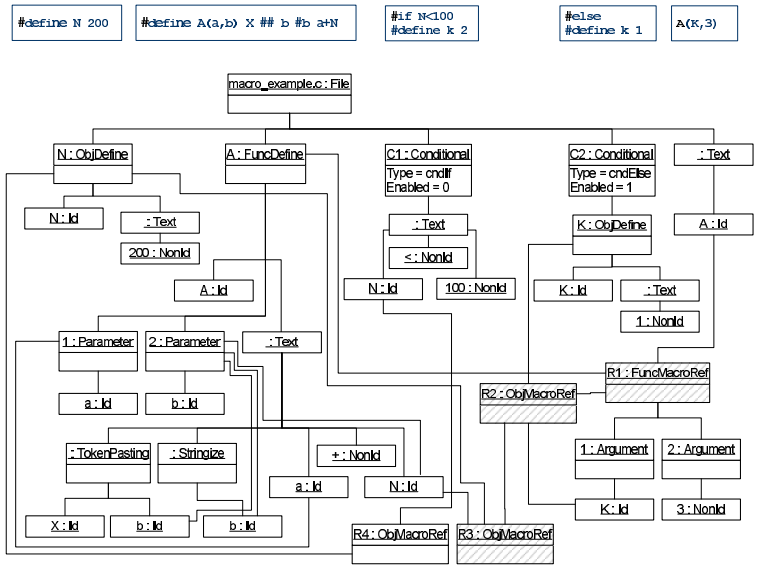

Fig. 6. Schema instance of a macro example

static information of the replacement list. The ***MacroRefs*** cannot be linked at the place of the definition as they belong to the place of the macro invocation. There is an `#if` conditional directive (`C1`) with a value of 0 (its body is ignored) and an `#else` pair (`C2`), which is preprocessed and contains the definition of macro `K`, given a value of 1. The last part of the code is the invocation of macro `A`. The ***FuncMacroRef*** object `R1` links the ***Id*** token with the actual arguments and the actual macro definition. The first parameter is a macro invocation (`K`) in this point of the code, so another macro reference is required (`R2`). The replacement list of macro `A` also contains a macro invocation (`N`) which is handled by `R3`. The three macro references are in the same reference class because they are used to describe the state of macro invocations at one place of the code (the corresponding objects are shaded in the diagram).

3.3 Using the Schema for information extracting

Both the original source code and the final preprocessed output can be extracted from a Schema instance. In addition, every intermediate state of the program which occurs during the transformations done by a preprocessor can be seen. Preorder tree walks on the generated tree are used to extract the requested information. The level of extracting and the kind of information is controlled by the depth and the details of the tree walk. Three scenarios are described here: getting back the original source after preprocessing, extracting the final preprocessed code and intermediate state investigation.

Original source. Retrieving the original source code during the preorder walk consists of the following actions. All text objects are written to output unchanged, including the `#define`, `#undef`, `#pragma`, `#error`, `#error` and `#` directives. Instead of included files only the `#include` directive is written out. After text elements with function-like replacement-references the argument list has to be written out. Conditional directives are also written out unchanged, but the finally retrieved source code is not the the same as the original source code, because the code from the false-evaluated conditional blocks are (currently) not restored.

Preprocessed file. Extracting the preprocessed file requires more operations during the walk. Included subtrees are written out, "silent" directives produce empty lines, the conditional blocks which has `true` value are written out. The macros are completely expanded with the help of ***MacroRef*** objects, the referenced ***Text*** objects are always replaced with their definition, recursively. The parameters are substituted and `#` and `##` operators are replaced by their resulting tokens.

Intermediate states. Analyzing intermediate states helps us better understand the job of the preprocessor. By an intermediate state of the source code we mean the level of macro expansion, and whether the included subtrees are placed instead of the directives. The latter is a trivial task during the tree walk.

Macro expansions are described with replace-references, `#` and `##` operators and parameter substitutions. When some of the macro strings are used and some of the replacement strings from active definition are used instead during the tree walk, then we get an intermediate state of source file (note that for a given class of ***MacroRef***s not all combinations are valid). Likewise we can choose whether to write out arguments with operators or write out the resulting token. Code may also be generated before and after a parameter substitution.

3.4 Environment of use and implementation

A reverse engineering tool called Columbus ([4], [3], [5]) has been developed in a cooperation with our department for some years now. Like most tools of this type, Columbus expects preprocessed input files. This approach bypasses analysing the code translations made by the preprocessor. Columbus is able to use any preprocessor, but currently the Microsoft *cl* [11], GNU *cpp* [15] and Borland *cpp32* are preconfigured. CANPP is a suitable alternative to these; we used *cl* and *cpp* for testing the tool on Win32 and Unix platforms, respectively. The CANPP tool has two tasks, namely to make an `.i` file as an ordinary preprocessor does, and to create Schema instances from the input. CANPP has now been integrated into Columbus and the Preprocessor Schema extends the Columbus Schema used by Columbus for additional analyses of whole programs.

The PCCTS [13] system was used to facilitate the lexical and syntactic analysis. We wrote an LL(1) grammar to syntactically analyse the input. The preprocessing is *syntax driven*, which means that as the parser recognizes a preprocessor directive then it calls the appropriate action. To evaluate integral constant expressions we created a separate parser.

4 Related work

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

In spite of its disadvantages, preprocessor directives are still widely employed. Ernst, Badros and Notkin [2] analysed the frequency and nature of preprocessor use in 2000. In their study they analyzed 26 commonly used Unix software packages written in C with about 970000 source lines (for example gcc, bash, emacs, gs, cvs...). Among other things they found that preprocessor directives make up the relatively high 8.4% of lines (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 [7] analysed the complex configuration structures created with directives and produced a graphical output of them. Latendresse [9] created a tool for finding the conditions needed for a particular source line to get through the conditional compilation.

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 being analysed occurs during the parsing. It has only a couple of limitations. However, every user has to write his own code to produce the analysed output. In [8] Kullbach and Riediger worked along similar lines to us. They divided the code into foldable and non-foldable segments. All directives are represented by this structure, which is integrated into a text editor and can visually help the programmer. But their schema does not permit external use. Livadas and Small [10] developed the Ghinsu program slicing tool. They focused on macro expansions, which are solved using different coordinate mappings between calls and definitions. They included the Ghinsu Preprocessor to produce extra code for the slicing tool to describe the macros.

Our work significantly differs from above-mentioned ones in that it is not just limited to special applications, but it may be used in many other tools like those mentioned above. The CANPP tool generates an ordinary `.i` file, hence it can be incorporated into the build process of existing projects. Furthermore, the use of a standard notation and technology (UML, XML) facilitates interoperability between software tools.

5 Conclusions and further work

In this work we introduced the Preprocessor Schema and the corresponding CANPP preprocessor tool. It extends the Columbus Schema, but can be employed as a stand-alone tool as well. We showed that there are a number of real problems which can be overcome with the help of our Schema. This works for several reasons. The structure of the schema follows the structure of C/C++ source code, and it is sufficiently general. It is not specifically designed for any particular field, but it also has the possibility of answering special questions about source code. Last, but not least, the XML output of an analysis can be used by anyone for further analysis, for example in source browsers, pretty printers and code-understanding tools.

In future research we would like to extend the use of conditional compilation and source configurations. We plan to handle them unconditionally and extract information regardless of the actual source configuration (e.g. macro values). We have also started work on designing a compact link between the Columbus Schema and the Preprocessor Schema. We would like to support new output formats (like HTML) so that, with the help of these, we can generate additional information for special applications extracted from the schema instances.

References

1. Badros, G. J., and Notkin, D. A framework for preprocessor-aware C source code analyses. *Software - Practice and Experience 30*, 8 (2000), 907–924.
2. Ernst, M. D., Badros, G. J., and Notkin, D. An empirical analysis of c preprocessor use. In *IEEE Transactions on Software Engineering* (Dec 2002), vol. 28.

3. Ferenc, R., Beszédes, A., Tarkiainen, M., and Gyimóthy, T. Columbus - reverse engineering tool and schema for c++. In *Proceedings of the 6th International Conference on Software Maintenance (ICSM 2002)* (Oct. 2002), IEEE Computer Society, pp. 172–181.
4. Ferenc, R., Magyar, F., Beszédes, A., Kiss, A., and Tarkiainen, M. Columbus – tool for reverse engineering large object oriented software systems. In *Proceedings of the Seventh Symposium on Programming Languages and Software Tools (SPLST 2001)* (June 2001), University of Szeged, pp. 16–27.
5. Homepage of FrontEndART Ltd. `http://www.frontendart.com`.
6. International Standards Organization. *Programming languages — C++*, ISO/IEC 14882:1998(E) ed., 1998.
7. Krone, M., and Snelting, G. On the inference of configuration structures from source code. In *Proceedings of the 16th International Conference on Software Engineering* (1994).
8. Kullbach, B., and Riediger, V. Folding: An Approach to Enable Program Understanding of Preprocessed Languages. Fachberichte Informatik 7–2001, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2001.
9. 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)* (March 2003), pp. 170–179.
10. Livadas, P. E., and Small, D. T. Understanding code containing preprocessor constructs. In *IEEE Third Workshop on Program Comprehension* (1994), pp. 89–97.
11. Microsoft Developer Network Online Library. `http://msdn.microsoft.com`.
12. Object Management Group Inc. *OMG Unified Modeling Language Specification*, version 1.3 ed., 1999.
13. Parr, T. J. *Language Translation Using PCCTS and C++, A Reference Guide*. Automata Publishing Company, 1993.
14. Spencer, H., and Collyer, G. #ifdef considered harmful, or portability experience with C News. In *Technical Conference* (June 1992), pp. 185–197.
15. Stallman, R. M., and Weinberg, Z. *The C Preprocessor, A GNU Manual*, 2001.
16. Stroustrup, B. *The C++ Programming Language*, Third ed. Addison-Wesley, 1997.