Giving Meaning to Macros

Christopher A. Mennie
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
camennie@cs.uwaterloo.ca

Charles L. A. Clarke
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
claclark@plg.math.uwaterloo.ca

Abstract

With the prevalence of legacy C/C++ code, it continually becomes more important to address the issues of readability and maintainability. Especially when considering the problems with refactoring or migrating C/C++ code do we see how important a role preprocessor directives play. It is partially because of these preprocessor directives that we find it extremely difficult to maintain our code.

We outline a method of fact extraction and manipulation to create a set of transformations which will remove preprocessor directives from the original source, converting them into regular C/C++ code with as few changes as possible, while maintaining readability in the code. In addition we briefly explore some of the subtle issues that can arise when migrating preprocessor directives. After discussing the general architecture of our test implementation, we look at some metrics gathered by running it on two software systems.

1. INTRODUCTION

The C/C++ preprocessor (CPP) is a macro processor whose language constructs are prevalent in C/C++ code, especially when looking at legacy systems. CPP implements a simple scheme which allows the user to substitute sections of text, conditionally include textual sections, or create strings from a given input text. Traditionally this input text is C/C++ code, but CPP may be seen as a language in its own right. An almost complete grammar of CPP can be found in Favre [1].

The output from CPP has all the preprocessor directives removed and is given to the C/C++ compiler to be compiled, leading us to say that C/C++ programs are actually written in two languages, CPP and C/C++. Code from both languages is almost always intermingled in the original (un-compiled/un-preprocessed) source code and it is worth noting that their scoping rules are significantly different. Scoping blocks in C are either at a global level, or within well defined blocks, delineated by braces. Alternatively, definitions in CPP are all declared at a global level and can only be brought out of scope explicitly, using the `#undef` preprocessor directive. These differences in scoping rules lead to not only intermingled code, but also overlapping block structures, respective to both languages.

For simplicity's sake we would like to deal with a single language, rather than a heterogeneous mixture of the two. A method is proposed herein to migrate preprocessor directives into regular C/C++ code. The intended results focus on trying to keep the transformed code as close as possible to the original, in terms of both meaning and readability.

While preprocessor macros were necessary to write C code, their use has largely become a supererogatory effort in C++. In "The Design and Evolution of C++" [2] Stroustrup states that use of the C preprocessor should be avoided and details the introduced C++ features to help do just that. We also see that refactoring C or C++ is made all the more difficult when macros are used [3]. Furthermore the task of migration for C or C++ code, is greatly hampered by the inclusion of macros [4].

Aside from manipulating source code, there is also the issue of fact extraction and visualisation. Most of the available fact extractors work with code that has already been run through CPP, and as such does not provide any representation of the original macros. Often this may not matter as most macro uses are for constant variables [5]. Still, one may be interested in the dependencies between macros and code, had such constants actually existed as variables. One may be misled from the visualised call graph due to the transitive property of macros being used as functions. A C/C++ function which uses a function-like macro will appear to directly call any methods the macro calls. It would be convenient to treat such macros like their C/C++ counterparts, in hopes of providing a clearer representation of the original source code.

1.1. Overview

The largest problems in dealing with things like migration or refactoring C/C++ code stems from the intermixing of the two languages. One solution would be to get rid of one of languages, namely CPP. Ideally in doing this, we would want to make as few changes as possible to the original source. CPP itself will translate the code into straight C/C++, of course, but we loose the implied semantic information that was encoded in the preprocessor directives not to mention a great deal of readability. We also want to preserve the original functionality as closely as we can, in that minor differences in performance or execution branches are acceptable so long as the program runs as it did before.

The approach we have taken here is to rewrite the original macros using C++ language constructs which mimic the macro as closely as possible. We outline a method to perform this task, and present some early results of its usefulness using our test implementation. While our aim is to handle as many types of macro use as possible, we can currently only migrate simple constant macros. It may seem fairly trivial to migrate such

simple constructs, but in truth a substantial amount of work was required. Fortunately the majority of this work applies to migrating most other types of macro use.

We know that macros are mostly used for simple tasks, like constants and inlined functions, and so it might beg the question as to why removing them would be difficult. While in truth most are not difficult to rewrite, the two main issues to deal with are the different language scoping rules and the lack of strong types for the macros.

Our approach is intended to be as straightforward and easy to perform as possible, however, as always the devil is in the details. We take the following steps in our method:

1) Extract the code and macro facts
2) Choose the order in which to migrate each macro
3) Determine how each macro is being used
4) Generate a plan to transform each macro
5) Transform each macro

It is intended to be as straightforward and intuitive as possible. We have found that the hardest part with this project was in discovering the more obscure language rules that our migration engine is likely to run into, like those discussed in section 3.2.

Macro use in C was typically for enhancing the C language, which has been largely dealt with in C++. As such, the focus of this paper will be on C++ and not C since there are far fewer options for the rewriting or migrating of macros in C. The method outlined is equally valid for C code, just with fewer migration options available.

1.2. Related Work

In conducting this research, we were inspired by the work of Ernst, Badros and Notkin [5], [6]. They examine preprocessor usage in 26 large C programs [5] and describe a tool that allows preprocessor constructs to be analyzed along with the remainder of the source in a unified framework [6]. Along with other applications, the tool was applied to replace macro definitions for constant values with equivalent C declarations [7]. Unfortunately, the structure of the tool restricts an analysis to a single file at a time, since it is based on existing compiler technology. The work was not extended to other macro types or to C++.

In this paper, we take a different approach to the problem, analyzing a complete software system before and after preprocessing, and merging the resulting factbases into a unified description of preprocessor transformations. This approach permits preprocessor constructs in header files to be translated into C++ constructs that reflect their usage throughout the system. While we focus on constant macros in this paper, our approach provides a framework for handling other preprocessor constructs and for accommodating different dialects of C++, including C.

Several other researchers have developed preprocessor-aware methods for analyzing C/C++ source [8], [9], [10]. The problem of tracking substitutions through the preprocessor is examined by Kullbach and Riediger [9]. Their *folding* method allows a user to visualize the actions of the preprocessor on a particular construct. Cox and Clarke [8] describe a technique for mapping facts, expressed as XML and generated by an analysis of the preprocessed source, back through the preprocessor to be properly situated in the original source. Malton et al. [11] describe a "source factoring" process that aids the analysis and transformation of code written in PL/1, and other languages, where preprocessor and macro constructs are heavily used.

Conditional compilation poses a particularly serious problem to a software analysis system. Text excluded by an `#ifdef` may contain syntax errors, code in a different dialect, comments, or complete gibberish. Given the problems they cause [12], source code should be re-written or transformed to eliminate these constructs, but in some cases this transformation may be impossible. Somé and Lethbridge [13] discuss many of the problems associated with conditional compilation and describe a parsing method for efficiently processing conditionally excluded code. Others [14], [15] have applied symbolic execution and partial evaluation techniques to analyze conditional constructs.

2. MACRO FACT EXTRACTION

Normally C/C++ fact extractors provide a representation of the code which closely resembles the abstract syntax graph (ASG) which a compiler compiling the code would create. However, for our purposes, we express the code in four different ways:

1) Original code, unprocessed by CPP or compiler
2) As facts representing the CPP language occurrences throughout the code
3) Code that has been processed by CPP, but not by the compiler
4) The compiler's ASG representation

If we realise that every CPP directive is expanded to at most a single line, we can safely ignore the third representation. This allows us to assume that line numbers which occur in the ASG directly correspond to the same line numbers in the original code, simplifying the analysis in our method somewhat.

2.1. Extracting macro facts

With the lack of macro fact extractors available it was necessary to write one. After some exploration, it became obvious that one would essentially need to write an entire CPP to extract macro facts. Rather than embark on this task, it was felt that an approach similar to the one taken with the CPPX [16] fact extractor should be chosen. Since CPPX was created as a patch to the GNU GCC C/C++ compiler, plus supporting tools, we wrote our macro fact extractor by modifying the GNU GCC CPP and creating the tool we called CPP0-CPPX. This gave us the power of a stable and well established CPP to work with.

CPP was modified to collect information about every preprocessor directive and macro expansion and to write these out into a file. The code fragment in figure 1 provides a simple example, which results in the set of facts shown in figure 2.

```
#define var 4
#ifdef var
#else
#endif
```

Figure 1: Simple code fragment with preprocessor directives

```
FACT TUPLE :
$INSTANCE @12 cMacroDecl
$INSTANCE @13 cDefinitionToken
$INSTANCE @17 cMacroIfdef
$INSTANCE @20 cMacroElse
$INSTANCE @23 cMacroEndif
cMacroDefines @13 @12
cMacroConditional @17 @12
cMacroBlockPart @17 @20
cMacroBlockPart @20 @23
FACT ATTRIBUTE :
@12 { name = "var" file = "test.c"
     line = 2 startLine = 1 }
@13 { type = CPP_NUMBER value = "4"
     file = "test.c" line = 1
     sourceColumn = 13 }
@17 { name = "var" file = "test.c"
     line = 3 entered = true }
@20 { file = "test.c" line = 4
     entered = false }
@23 { file = "test.c" line = 5 }
```

Figure 2: Example macro facts in TA format

Like CPPX, we represent our macro facts in the TA format [17]. This scheme allows us to represent the facts and their relations as a graph. Each node in the graph is given a type through the `$INSTANCE` declaration and associated attributes in the `FACT ATTRIBUTE` section. The inter-node relations are described in the remaining lines of the `FACT TUPLE` section.

We can see from the example that each macro declaration includes facts about its expansion. More complex preprocessor code would provide the string of token expansions, as well as facts about parameter expansion occurrences for parameterized macros.

Generally speaking, extracting the macro facts is straightforward, however there were some subtleties to overcome. From figure 2 we note that each "part" of a preprocessor conditional is declared as a fact and their association strung together. It is also necessary to keep track of whether or not a particular block was actually included in the output code. As CPP processed directives within each block, irregardless of if they were included in the output code or not, we had to keep track and ensure that if they were in a "skipped" block then no errant directive facts were placed in the resulting fact file.

Another source of frustration was in dealing with macros whose declarations spanned multiple lines. As far as CPP is concerned, every declared macro is a single line. A multi-lined macro declaration must therefore be written with the line continuation marker "\". However, at least with the GNU GCC CPP, the internals consider the line being tokenized to remain the same throughout the multi-lined macro declaration. What this results in is a set of facts which appear to interlace the declaration tokens and provide generally nonsensical results. One needs a way of knowing the real line, relative to the original source, a given expansion token occurred on as well as what line CPP believes it to be on. This point is clarified in figure 3 with corresponding facts shown in figure 4.

```
#define var 1\
            2
```

Figure 3: Simple code fragment with multi-line macro declaration

```
FACT TUPLE :
$INSTANCE @12 cMacroDecl
$INSTANCE @13 cMacroDefinitionToken
$INSTANCE @15 cMacroDefinitionToken
AndThen @13 @15
cMacroDefines @13 @12
FACT ATTRIBUTE :
@12 { name = "var" file = "test5.c"
     line = 3 startLine = 1 }
@13 { type = CPP_NUMBER value = "31"
     file = "test5.c" line = 1
     sourceColumn = 13 }
@15 { type = CPP_NUMBER value = "32"
     file = "test5.c" line = 2
     sourceColumn = 3 origMultiLine = 1
```

Figure 4: Example multi-line macro facts in TA format

One final obstacle to overcome was the issue of maintaining consistent file paths in the fact file. When a file is included through the `#include` directive it is possible that it existed in the same path as the including file. Alternatively, it is possible that it was in a different directory and the include directive used a relative or absolute path. This becomes an issue when the same file is included multiple times, from different files, by different path names. If one is not careful, the resulting fact file could contain multiple references to the same file, but with different path names. We decided that every included file should be referenced in the fact file by either an absolute path or a path relative to the original C/C++ source code being compiled. This also is consistent with the extracted fact output from CPPX.

2.2. Extracting facts from original code

To extract facts about the original code it was felt the best approach was to tokenize the code and record the important attributes for each token, as exemplified in figures 5 and 6. To be consistent with CPPX and CPP0-CPPX we again used TA

```
a = 16;
```

Figure 5: Simple code fragment

```
FACT TUPLE :
$INSTANCE @8 cDefinitionToken
$INSTANCE @9 cDefinitionToken
$INSTANCE @10 cDefinitionToken
$INSTANCE @11 cDefinitionToken
AndThen @8 @9
AndThen @9 @10
AndThen @10 @11
FACT ATTRIBUTE :
@8 { type = CPP_NAME value = "a"
     file = "test.c" line = 1
     sourceColumn = 1 }
@9 { type = CPP_EQ value = "="
     file = "test.c" line = 1
     sourceColumn = 3 }
@10 { type = CPP_NUMBER value = "16"
     file = "test.c" line = 1
     sourceColumn = 5 }
@11 { type = CPP_SEMICOLON value = ";"
     file = "test.c" line = 1
     sourceColumn = 7 }
```

Figure 6: Example token facts in TA format

as the output format.

The naming convention used for the token types is the same as the one used by GCC's CPP. Since CPP must tokenize the code it is processing, we modified it to not only output macro related facts, but tokenized code facts as well.

3. Approach to Migrating Macros

Fact Extraction

As outlined in section 1.1, we take a straightforward approach to the migration of macros. The first step is to extract the facts about the system. For each compiled source we use CPP0-CPPX to extract the macro and lexer token facts, and CPPX to extract the ASG facts. With these three separate fact files, we ensure that each node has a unique name and then merge the three fact bases together. With this new fact base we remove any of the facts that we will not be needing; the standard C include header facts in particular. From here we merge the facts into the overall project fact base. Finally duplicate entries in the fact base are identified, as they may have different names but the same attributes, and removed.

Once we are satisfied with our extracted facts, we choose the order in which to migrate the macro instances. This is largely intended for macros defined via a `#define` preprocessor directive, although it may matter for removing `#if` statements and the like. Since we are not extracting facts from the output of CPP, we do not want to expand macros out in our migration

```
#define A 6
#define B A+7
int func() {
     return B;
}
```

Figure 7: Example of indirect expansion of macro A from direct expansion of macro B

```
#if defined(UNIX) && !defined(SMALL)
     && defined(CONFIGURABLE)
```

Figure 8: UNIX, SMALL, and CONFIGURABLE can be seen as being co-dependent on one another

system. Ideally we want to only need to know where each macro is directly expanded, in the original source code, rather than indirectly expanded, as seen in figure 7.

Due to this, we need to isolate groups of macros which must be migrated at the same time to ensure the correctness of the resulting migrated code. The easiest to choose are the macros which do not make use of other macros, as opposed to possible groups of macros which depend on each other. Depending on how one handles `#if` statements, one could deem all macros in an `#if` statement condition as being dependent on each other, like in figure 8.

Classification

After deciding the order in which to migrate the macros, we determine as much as we can from each macro in the prescribed sequence. This involves looking at how each macro is used and assigning a classification, of which we have identified almost two dozen types of macro classifications, which are briefly described in section 3.1. The macro being processed is seen if it matches the criteria for each classification type. If more than one classification criteria is met, then a resolution heuristic is applied to determine which classification best matches the macro's use. We note that multiple classifications mean that any of them would be technically correct, in that the resulting transformations would work. The heuristically chosen match is intended to choose the classification which mirrors as closely as possible the intent the original coder had when creating the macro (figure 9).

To optimize the classification process we have allowed for multiple passes to be performed. Certain classifications are subclassifications of others and rather than eliminating these possibilities as quickly as possible, we wait until the resolution

Code	**Classifications for A**	**Resolution for A**

`#define A 5`
`#ifdef A`
`#endif`
	1) Constant Value
	2) Configuration Setting
		Constant Value

Figure 9: Example of classification resolution

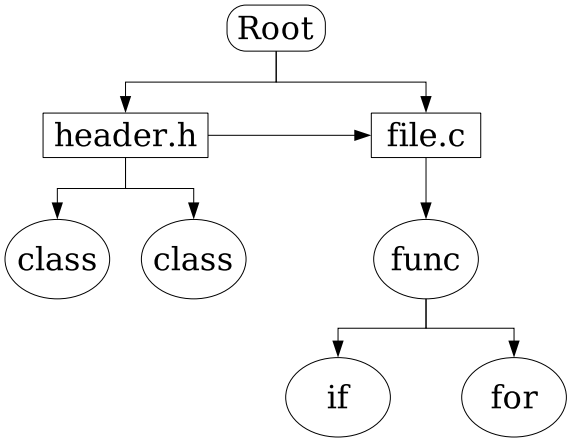

Figure 10: Block dependency graph example

Ex 1

```
int funcA()
{
     #define A 5
     return A;
}

int funcB()
{
     return A;
}
```

Ex 2

```
#define A 5

int func()
{
     if(true) {
          return A;
     }
     if(true) {
          return A;
     }
}
```

Figure 11: Examples of:
1) Overlapping scope
2) Original macro declaration as ancestor to LCA of uses

of the previous pass has been completed. So far only two passes have been found necessary, but the possibility for more remains.

From the classification of the macro we determine the transformations necessary to migrate the macro. For the purposes of declarative macros, the two main things we need to determine are the new scope of the migrated macro and what C type to associate with it.

To deal with the scoping issue, we create a directed graph representing the C blocks found in the system and their respective types, say a function or `for` block. We represent an included file as being the ancestor node of the including file. An example of the block dependency graph structure can be seen in figure 10.

Inferring a type for the macro involves delving into the fact base and seeing how the compiler made use of the given macro.

Placement

Once we have this block dependency graph, we look for each block in which the macro to be migrated has been used, outside of a preprocessor directive. The least common ancestor of these blocks in the graph then represents a block in which the resulting migrated declaration of the macro may be placed. Due to the different scoping rules between C/C++ and CPP, we need to locate this block in case the scope of the macro is not consistent between the two languages. In an attempt to minimise the structural changes to the code a rule was added that states if the block in which the macro is declared has a path in the block graph to the discovered least common ancestor block, then the resulting block in which the migrated macro declaration will be placed, will be the same as that of the original declaration. In other words, just because we could declare the migrated macro "closer" to where its been used, does not mean we should necessarily do so. The original declaration tends to be in a reasonable place, like a header file for example, and should remain there if possible (figure 11).

Legal

```
switch(something) {
case 4:
{
     int macro = 7;
}
}
```

Illegal

```
switch(something) {
case 4:
     int macro = 7;
}
```

Figure 12: Switch declaration examples

Though we know in which block we want the migrated macro declaration to be in, we still have to choose a specific line to place it on. If the original declaration was in the target block, then we simply try to replace the old declaration with the new. Otherwise we choose the line before the first use of the macro or as close as possible if the target block is different from the first use. In some cases it may not be possible to insert a new declaration at those initially chosen lines. The target declaration line, for example, may be in the middle of a statement that has been split across multiple lines. Or the target block can not include the type of declaration we wish to make. An example of the latter case is shown in figure 12. We can not declare a variable within a switch statement, unless the particular case label contains a proper C block, braces and all.

To resolve the above issue we start from the target line in the target block, and search backwards in the file until we reach the first line which will legally accept the migrated macro declaration. After all this work we have one final task to perform, in that we must check for any name conflicts which may now arise. As we have possibly moved the declaration to a new spot in the source, we are now at risk of causing a name conflict with other entities in the code. To resolve any conflicts we must decide whether to rename either the migrated macro, or that which it conflicts with.

```
<MacroInfo>
<File fileName="keymap.h">
     <DelLine fileName="keymap.h"
          lineNumber="24"/>
     <InsertLine fileName="keymap.h"
          lineNumber="25"
          lineVal="const int KOF = 0;"/>
</File>
</MacroInfo>
```

Figure 13: Example transformation file

Transformations

Once the renaming issue is resolved, we take all this information gathered about the macro and create a series of transformation steps. We define a set of simple commands to insert or delete a line or lexer token based on the information gathered, like those shown in figure 13. After all the macros have been processed, this allows us to form a series of steps in which if followed will take the original source and transform it with the newly migrated macros.

We use a simple script that takes the transformations to be performed and actually performs them on the source code.

All that said and done, we accept that there may be instances of macro usages that simply can not be migrated without extensive code rewriting, best left to the software engineers.

3.1. Macro Classifications

We have classified macro use into almost two dozen types. Our taxonomy is inspired by the styles of macro use that we've encountered in the systems used for our case study (section 5). Below we present a brief description of each of these classifications:

Unparameterized Macros

Simple Constants: The body of these macros expand out to a single constant value. This value may be a numeric value, string, or character. Expansions that involve negative values or extraneous paired parenthetical marks are included in this classification.

```
#define VALUE 5
#define WELCOME_MSG "Hello!"
#define LENGTH ((-(((53)))))
```

Constant Expression: Like simple constant macros, these macros expand out to a constant value. However, their body may contain arithmetic expressions or type casts, so long as the evaluated value is constant. Other variables may also be used in the expansion so long as they are constant values and for every expansion the same variables are used.

```
#define FIVE 3+2
#define FIVE (((3))+((1)+1))
#define UHELLO (unsigned char *)"Hello!"
#define SUMAB A+B /*A and B are
                    constant values*/
```

Function Alias: These parameterless macros contain a single word expansion, providing an alias for an already existing function. A practical use would be to conditionally define an alias for two or more similar functions.

```
#ifdef USE64BITS
#define SUM Sum64
#else
#define SUM Sum32
#endif
```

Type Alias: The body of these macros expand out to some C type or typedef'd type. A practical use would be to provide a common type with an alias to handle portability issues.

```
#ifdef USE_WIDE_CHAR
#define CHAR wchar
#else
#define CHAR char
#endif
```

Keyword Alias: The macro expands out to a C/C++ keyword.

```
#define CURRENT this
#define E extern
```

Variable Alias: These macros provide an alias to a global or local variable.

```
int CurrentBuildingHeight = 5;
#define HEIGHT CurrentBuildingHeight
```

Keyword Redefinition: The name for this kind of macro is a valid C/C++ keyword, with an expansion of either another C/C++ keyword or a semantically valid expression relative to the macro name.

```
#define void int
typedef long ulong
#define int ulong
```

Parameterless Function: These macros mimic parameterless C++ inlined functions. The macro expansion may use global variables so long as they are always the same variable.

```
int Score = 23;
#define TwiceScore (Score * 2)
#define Silly {int x=5; x=x+2;}
```

Parameterless Function With Variable Use: Unlike the parameterless function macros, the body of these macro expansions make use of local variables. Thus every invocation of the macro may use different variables, despite being similarly named.

```
/*Code snippet from functions A, B, and
C*/
int Score = 2; /*Not global*/
#define DoubleScore Score = Score << 2
```

Empty Declaration: The body of these macros are simply empty. As an example, we might use such a macro to cope with compilers that don't support certain keywords.

```
#define static
#define extern
```

Code Snippet: These macros don't expand out to well formed C/C++ expressions, but instead their expansions are snippets of code.

```
#define ENDIT return 0; }
```

Parameterized Macros

Inlined Function: These macros mimic C++ inlined functions that use parameters. The macro expansion may use global variables so long as they are always the same variable.

```
#define TwiceScore(Score) ((Score) * 2)
#define SillySum(X,Y) (X+Y+1)
```

Inlined Function With Variable Use: Unlike the inlined function macros, the body of these macro expansions make use of local variables. Thus every invocation of the macro may use different variables, despite being similarly named.

```
/*Code snippet from functions A, B, and
C*/
int Score = 2; /*Not global*/
#define AlterScore(X) Score = Score << X
```

Function Alias: The expansion of these macros provides an alias for an already existing function. A practical use would be to conditionally define an alias for two or more similar functions.

```
#ifdef FILEPRINT
#define PRINTSTR(STR) fprintf(TheFile,
STR)
#else
#define PRINTSTR(STR) printf(STR)
#endif
```

Parameterized Code Snippet: These macros don't expand out to well formed C/C++ expressions, but instead their expansions are snippets of code.

```
#define ENDIT(val) return val; }
```

Filter Tuple: These macros take a number of parameters and "filter" some of them out.

```
#ifdef DOFILTER
#define A(x,y,z) x
#else
#define A(x,y,z) {x,y,z}
#endif
```

Conditionals

Excluded Code: This conditional directive always excludes a code segment.

```
#if 0
...
#endif
```

Expressly Included Code: This conditional directive always includes a code segment. This may also be used to temporarily include a segment which would otherwise be classified as a configuration segment.

```
#if 1 && VERSION > 5
...
#endif
```

Configuration Setting: These conditional directives do not always include their respective code segments, but instead operate depending on defined macro values at compile time.

```
#if VERSION > 5 && defined(SOMEMACRO)
...
#endif
```

Header Sentinel: A common technique to ensure a header is only included once is to encapsulate it with an existential conditional.

```
/*Beginning of header*/
#ifndef HEADER_H
#define HEADER_H
...
#endif
/*End of header*/
```

Other

Token Pasting: These macros concatenate two tokens together.

```
#define MAKETYPE(type) typedef int
type##_type
MAKETYPE(int)
```

Literal Expansions: These macros make use of the literal string which represents their parameter(s).

```
#define ASSERT(EXP) printf(#EXP)
ASSERT(1+3=5);
```

3.2. Pitfalls in Migration

The approach laid out in section 3 generally works very well. For the most part, once the scoping and name conflict issues are resolved, the macro transformations immediately follow. However, we must also consider the semantics of the C language and the interesting situations which arise that are not immediately obvious, plus the potential varied use of a given macro.

3.2.1. Language Issues: One good example of a migration pitfall is when a macro is used inside a case label. Case labels must have constant values associated with them, and once we migrate a macro to a variable (constant or not), or method, the original code will no longer compile. Furthermore, in the case of migrating to a method, it is impossible to get the code to

compile without some significant changes and work-arounds, at the detriment of readability and maintainability.

Ultimately in this case one must change the code in a less than straightforward way to work around this problem. Our approach in handling simple constant macros which appear in case labels is to declare them, the macros, within an enumeration as opposed to a variable. We work with the assumption that this is what the original coder intended by the macro and group the enumerated migrated macros together. When multiple enumerations are to be made in the same block we look at which ones are used in the same switch statements and place them in the same enumeration, grouping them together in a greedy fashion.

A different problem is the use of gotos. When a macro is declared between the goto statement and the target label. Without taking this into consideration, the approach above could decide that the newly migrated declaration should simply reconstitute the one in place. However the goto jump will now cross over the new initialisation, causing a compilation error. We do not have this problem if the declaration is within a proper C block, of course.

Considering the maintainability aspects, we must also consider the issue of comments. While beyond the scope of this paper, if a migrated macro is placed in a different location, then any comments associated with it must also be similarly moved and placed.

3.2.2. Varied Macro Uses: An issue that has not been dealt with is what to do with macros whose use differs based on the context. More concretely, consider the case of when a macro's expansion includes the expansion of another macro. So long as both these macros are always in scope for every use, there is no problem. Should the macro within the expansion not be in scope some of the time it is used, relative to the encapsulating macro, then we have a problem as this implies that the encapsulating macro is sometimes using C variables that happen to be in scope when the macro is used.

Similarly for macros that are used in a polymorphic way, like templates for abstract data types, we again have this problem of the macro having different semantics depending on where it is used. Although we have not yet implemented these transformations, it is expected that some instances can be successfully migrated into C++ templates. In other cases however, we could duplicate the original macro based on its different instances of use, but this leads to readability/maintainability problems and naming issues.

4. IMPLEMENTATION

Our implementation is made up of three phases. First there is the fact extraction phase, then the migration engine which determines all the transformation actions to take. Finally there is the source transformation phase which takes the original source and actually performs the prescribed transformations upon it.

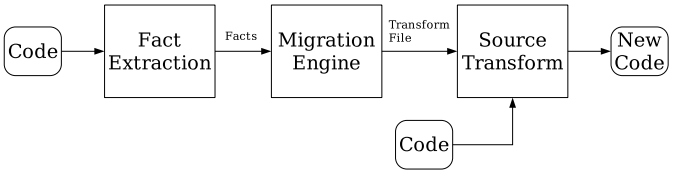

Figure 14: Overall implementation architecture

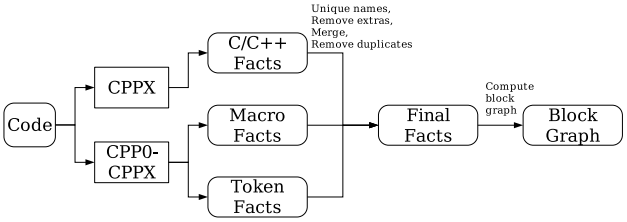

Figure 15: Fact extraction architecture

4.1. Fact Extraction

Our fact extraction process is not as straightforward as it tends to be with other systems. We begin by using CPPX and CPP0-CPPX to extract the C/C++, macro, and source code token facts. Remembering that we are going to repeat this for every compiled source we take the generated fact files and perform some prep work on them.

First we ensure each node in the fact file has a unique name by appending the filename of the original source to the name of each node. Once that is done, we merge the three fact files together. From there, all references to files we are not interested in are removed. This is accomplished by maintaining a list of the source files to include, and simply removing everything else. We then add these facts to the entire system's fact base. In all likelihood we will have added duplicate facts and so we look for nodes whose type and attributes are the same, removing the duplicates, and renaming any old references to the remaining node.

Most of the manipulation of the fact files is done using a relational database called Grok [18], with the node duplicate removal being done through a tool of our own.

After the system has gone through all the compiled source files and built up our fact base for the system, we finish the process by creating a block dependency graph. We do this by first finding where each block lies in the source, from every pair of "curly" braces. Every block found is then examined through the facts which relate to its respective beginning line number, and from that we infer the type of block. The block graph is then stored in a separate file.

4.2. Core Migration Engine

The heart of the implementation is contained within this component. In here we actually determine the type of each macro and how to transform them.

We start with the fact base and block graph; deciding the order in which to process the macros. As we currently only

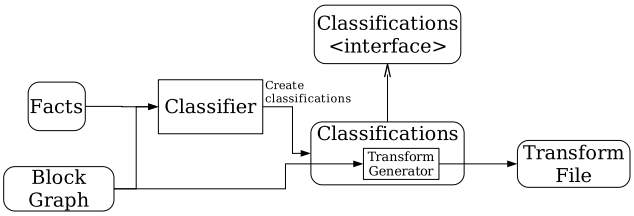

Figure 16: Core migration engine architecture

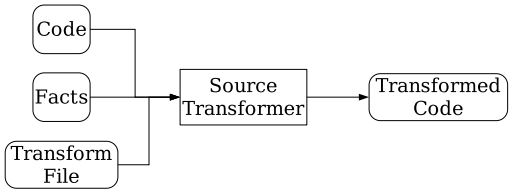

Figure 17: Source transformer architecture

handle simple constant macros, this decision is fairly easy. Ultimately though, we will need to construct a graph of macro dependencies in the fact extraction stage. With that graph we would find small subgraphs of dependent macros to process at the same time, in that they must all be processed for the resulting transformations to give valid and compilable results.

For each macro we determine the type of classification to apply to it. Given the macro node name, we ask our system if it matches each of the classification types. This is done by creating a generic list of all the classification types (represented as flyweight objects), and asking each in turn if the macro fits its classification criteria. We then take the list of matched criteria and resolve which one would be the "best fit" for the macro. Currently this is an easy task as we have so far implemented the handling of a single classification type. As more classification types are added, the resolution heuristic will complicate accordingly.

One of the intended consequences of using this generic means of managing classification objects, is the ability to use different versions of the same classification. A good use of this would be to select the flavour of C/C++ that we target. Transformations that are perfectly valid in C++ would not necessarily be valid in C99, for instance. Alternatively there are things that the GNU GCC compiler will do that other compilers can not, and our modularity with the classification objects allows us the flexibility to choose our target platforms.

With the macro classification in hand, as an object, we generate the transformations necessary to migrate the original macro. Depending on the type of classification applied to the macro, the necessary information required to migrate it may be discovered either in the classification stage or the transformation gathering stage. For example, simple constants have their expansions recorded, type of variable to create, and whether or not its used in a case label, all gathered during the classification stage. Other information regarding placement of the migrated macro is determined during the transformation stage.

We represent the transformations as a generic list of transformation objects, sorted by file and line. When finished processing, an XML file containing the transformations is written.

4.3. Source Transformer

Once we have generated all the transformation steps necessary to migrate the macros of a given system, we then proceed with performing the source code transformations.

5. Case Study

To demonstrate the usefulness of our system, we chose one small and one medium sized program as test cases for our implementation. Chosen were the text editor Vim, version 3.0, and the eternal game of Nethack, version 3.4.0. Both programs were built using a standard configuration for x86 Linux systems. Our results covered both sources and headers.

Our goal was to see just how many simple constant macros required more than a straightforward line replacement to migrate them. We found that very few of the macros caused a naming conflict relative to their target location of migration. All of the name conflicts were the result of redeclaring a macro of the same name, with the same value, revealing some probable code rot.

We did not expect that many migrated macro declarations would need to have been moved to a different scoping block. This held true for Nethack, but Vim required a surprisingly high number of moves.

Our expectations for the number of simple constant macros used in the programs was roughly met. Both programs have a large lists of sequential values representing commands, monsters, etc. Not surprisingly, these were used in switch statements.

The results are summarized in table 1.

Metric	Vim	Nethack
# of Files	51	195
KLoC	26	108
Total # of declared macros	388	3625
Total # of simple constant macros	263	2465
# required moving	148	78
# used in case labels	166	644
# causing name conflict	2	4

TABLE 1: Results of experiment on Vim and Nethack

6. Conclusions

We've explored within this paper a unique approach to migrating C preprocessor directives into C/C++ code, while focusing on readability and maintainability. This approach involved extracting facts about the macros in the system, determining how each macro is being used, and then generating a set of transformations in order to migrate the macros into C/C++ code.

From the substantial work put into the implementation, we describe its operation and capabilities, as well as the next steps in its evolution. To demonstrate the value of our implementation, the code for two applications was processed with our system and some metrics on the resulting transformations recorded.

We know that our implementation is currently able to migrate a significant number of macros in the average application. With some more work we intend to increase the migration rate to the large majority of macro instances within an average application.

REFERENCES

[1] J.-M. Favre, "The CPP Paradox," *http://citeseer.nj.nec.com/favre95cpp.html*, 1995.

[2] B. Stroustrup, *The Design and Evolution of C++*. ACM Press/Addison-Wesley Publishing Co., New York, NY, 1995.

[3] A. Garrido and R. Johnson, "Challenges of Refactoring C Programs," in *Proceedings of the International Workshop on Principles of Software Evolution*, pp. 6–14, ACM Press, New York, NY, 2002.

[4] J. Martin, *A C to Java Migration Environment*. PhD thesis, University of Victoria, 1996.

[5] M. D. Ernst, G. J. Badros, and D. Notkin, "An Empirical Analysis of C Preprocessor Use," *IEEE Transactions on Software Engineering*, vol. 28, pp. 1146–1170, December 2002.

[6] G. J. Badros and D. Notkin, "A Framework for Preprocessor-Aware C Source Code Analysis," Technical Report UW-CSE-98-08-04, Computer Science and Engineering, University of Washington, 1999.

[7] G. J. Badros, "PCp^3: A C Front End for Preprocessor Analysis and Transformation." *http://www.cs.washington.edu/homes/gjb/papers/constraints-iga.pdf, 1997*.

[8] A. Cox and C. Clarke, "Relocating XML Elements from Preprocessed to Unprocessed Code," in 10th International Workshop on Program Comprehension, *(Paris), pp. 229–238, 2002.*

[9] B. Kullbach and V. Riediger, "Folding: An Approach to Support Understanding of Preprocessed Languages," in 8th Working Conference on Reverse Engineering, *(Stuttgart, Germany), pp. 3–12, 2001.*

[10] P. Livadas and D. Small, "Understanding Code Containing Preprocessor Constructs," in 3rd International Workshop on Program Comprehension, *(Washington D.C.), pp. 89–97, 1994.*

[11] A. Malton, J. R. Cordy, D. Cousineau, K. A. Schneider, T. R. Dean, and J. Reynolds, "Process Software Source Text in Automated Design Recovery and Transformation," in 9th International Workshop on Program Comprehension, *(Toronto, Canada), 2001.*

[12] H. Spencer and G. Collyer, "`#ifdef` *considered harmful, or Portability Experience with C News," in* USENIX Summer Conference, *(San Antonio, Texas), pp. 118–125, 1992.*

[13] S. Somé and T. Lethbridge, "Parsing Minimization when Extracting Information from Code in the Presence of Conditional Compilation," in 6th International Workshop on Program Comprehension, *(Ischia, Italy), 1998.*

[14] Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe, "C/C++ Conditional Compilation Analysis using Symbolic Execution," in International Conference on Software Maintenance, *(San Jose, California), pp. 196–206, 2000.*

[15] I. Baxter and M. Mehlich, "Software Change Through Design Maintenance," in International Conference on Software Maintenance, *(Bari, Italy), pp. 250–259, 1997.*

[16] R. Holt, T. Dean, and A. Malton, "CPPX - C/C++ Fact Extractor CPPX," http://swag.uwaterloo.ca/~cppx/, *January 2004.*

[17] R. Holt, "TA: The Tuple Attribute Language," http://plg.uwaterloo.ca/~holt/papers/ta-intro.htm, *2002.*

[18] R. Holt, "Introduction to the Grok Language," http://plg.uwaterloo.ca/~holt/papers/grok-intro.html, *2002.*