# Giving Meaning to Macros

by

Christopher Allan Mennie

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2004

©Christopher Allan Mennie 2004

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

# Abstract

With the prevalence of legacy C/C++ code, issues of readability and maintainability have become increasingly important. When we consider the problem of refactoring or migrating C/C++ code, we see the significant role that preprocessor directives play. It is partially because of these preprocessor directives that code maintenance has become extremely difficult.

This thesis describes a method of fact extraction and code 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, some of the subtle issues that may arise when migrating preprocessor directives are explored. After discussing the general architecture of the test implementation, an examination of some metrics gathered by running it on two software systems is given.

# Contents


**1 Introduction** **1**

1.1 Motivation . . . 1

1.2 Objective . . . 5

1.3 Thesis Overview . . . 7

**2 Background and Related Work** **8**

2.1 The C/C++ Preprocessor . . . 8

2.1.1 `#include` Directive . . . 8

2.1.2 Conditional Directives . . . 9

2.1.3 Macro Expansion . . . 11

2.1.4 Other CPP Features . . . 14

2.2 Fact Extractors . . . 15

2.2.1 CPPX . . . 17

2.3 TA Modelling Language . . . 20

2.3.1 TA Macro Schema . . . 21

2.4 Grok . . . 28

2.4.1 Language . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30
2.4.2 Embedded Grok Interpreter . . . . . . . . . . . . . . . . . . . . . 31
2.5 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

**3 Approach to Migrating Macros 34**

3.1 Fact Extraction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 34
3.2 Classification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
3.3 Placement . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.4 Transformations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.5 Pitfalls in Migration . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.5.1 Language Issues . . . . . . . . . . . . . . . . . . . . . . . . . . . 41
3.5.2 Varied Macro Uses . . . . . . . . . . . . . . . . . . . . . . . . . . 43

**4 Macro Fact Extraction and Classification 44**

4.1 Macro Fact Extraction . . . . . . . . . . . . . . . . . . . . . . . . . . . 44
4.1.1 Extracting macro facts . . . . . . . . . . . . . . . . . . . . . . . . 45
4.1.2 Extracting facts from original code . . . . . . . . . . . . . . . . . . 48
4.2 Macro Classifications . . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.1 Unparameterized Macros . . . . . . . . . . . . . . . . . . . . . . . 50
4.2.2 Parameterized Macros . . . . . . . . . . . . . . . . . . . . . . . . 55
4.2.3 Conditionals . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 56
4.2.4 Other . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
4.3 Macro Migrations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59
4.3.1 Implemented Migrations . . . . . . . . . . . . . . . . . . . . . . . 59

4.3.2 Possible Future Migrations . . . . . . . . . . . . . . . . . . . . . . . . . . 61
4.3.3 Current Unknown Migrations . . . . . . . . . . . . . . . . . . . . . . . . 65

**5 Implemented Architecture and Framework** **66**
5.1 Document Conventions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.2 Fact Extractor . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.3 Migration Engine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.3.1 Driver . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
5.3.2 Coordinator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
5.3.3 Transformation Store . . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.3.4 Second Stage Singletons . . . . . . . . . . . . . . . . . . . . . . . . . 72
5.3.5 Classification Worker . . . . . . . . . . . . . . . . . . . . . . . . . . 73
5.3.6 Classification Type . . . . . . . . . . . . . . . . . . . . . . . . . . . 77
5.4 Source Transformer . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
5.5 Process View of Architecture . . . . . . . . . . . . . . . . . . . . . . . . . 80

**6 Experimental Results** **83**
6.1 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 84
6.2 Confessions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
6.2.1 Nethack . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
6.2.2 Vim . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

**7 Conclusion** **92**
7.1 Future Work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93

**Bibliography** **94**

# List of Tables

6.1 Results of experiment on Vim and Nethack . . . . . . . . . . . . . . . . . . . . . . 84
6.2 Explanation of metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
6.3 Details for simple constant macros . . . . . . . . . . . . . . . . . . . . . . . . . . 89
6.4 Details for enumerated constant macros . . . . . . . . . . . . . . . . . . . . . . . 90
6.5 Details for constant expression macros . . . . . . . . . . . . . . . . . . . . . . . . 90
6.6 Details for parameterless function macros . . . . . . . . . . . . . . . . . . . . . . 90
6.7 Details for keyword alias macros . . . . . . . . . . . . . . . . . . . . . . . . . . . 90
6.8 Details for empty declaration macros . . . . . . . . . . . . . . . . . . . . . . . . . 91

# List of Figures


1.1 Example code with preprocessor directives . . . . . . . . . . . . . . . . . . . . . . . . 3

1.2 Example code with preprocessor directives migrated away . . . . . . . . . . . . . . 4

2.1 Simple `#include` directive examples . . . . . . . . . . . . . . . . . . . . . . . . 9

2.2 CPP Conditional Examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.3 Migrated CPP Conditional Examples . . . . . . . . . . . . . . . . . . . . . . . . 11

2.4 Example facts of a `for` loop within a function in TA format . . . . . . . . . . . . 17

2.5 Example literal source code facts . . . . . . . . . . . . . . . . . . . . . . . . . . 18

2.6 Example of preprocessor operation facts . . . . . . . . . . . . . . . . . . . . . . . 19

2.7 Complete TA example . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21

3.1 Example of indirect expansion of macro A from direct expansion of macro B . . . 35

3.2 UNIX, SMALL, and CONFIGURABLE can be seen as being dependent on one another . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.3 Example of classification resolution . . . . . . . . . . . . . . . . . . . . . . . . . 36

3.4 Block dependency graph example . . . . . . . . . . . . . . . . . . . . . . . . . . 37

3.5 Examples of overlapping scope and original declaration as ancestor to LCA of uses 39

3.6 Switch declaration examples . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

3.7 Example transformation file . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
3.8 Un-realizable migration examples . . . . . . . . . . . . . . . . . . . . . . . . . . 42

4.1 Simple code fragment with preprocessor directives . . . . . . . . . . . . . . . . . . 45
4.2 Example macro facts in TA format . . . . . . . . . . . . . . . . . . . . . . . . . . 46
4.3 Simple code fragment with multi-line macro declaration . . . . . . . . . . . . . . . 47
4.4 Example multi-line macro facts in TA format . . . . . . . . . . . . . . . . . . . . . 48
4.5 Simple code fragment . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
4.6 Example token facts in TA format . . . . . . . . . . . . . . . . . . . . . . . . . . 49

5.1 High Level Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
5.2 Fact Extractor Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
5.3 Migration Engine Component . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
5.4 Classification Worker Component . . . . . . . . . . . . . . . . . . . . . . . . . . 74
5.5 Classification Type Component . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
5.6 Source Transformer Component . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
5.7 Migration Engine Process View . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

# Acknowledgements

First and foremost I would like to thank my supervisor, Charlie Clarke. I would like to say thanks for accepting me as a student despite my marks, and allowing me the autonomy to follow my own research interests. If it were not for Charlie pushing me to submit a conference paper, I might never have had the wonderful opportunity of backpacking in Europe.

I would like to dedicate this thesis to my daughter, Harmony. You have been a source of great joy and inspiration, and I love you very dearly.

Finally, I would like to thank my friends and family. Your help and guidance has been much appreciated.

Kallisti.

# Chapter 1

# Introduction

## 1.1 Motivation

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 [6].

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, a single language would be easier to deal with 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. The term "migrate" has been chosen to convey the process because in essence this method is migrating one language, CPP, into another, that being C/C++. While the migration method is somewhat unorthodox, in that it requires the "outside" C/C++ code that the CPP code expands into, the term "migration" is still felt to be appropriate.

As an example, if we consider the code in figure 1.1, despite the use of preprocessor directives we should have an intuitive feel for what those directives are really representing. The simple constant macros are just static constants, whereas the parameterized macros are acting as inline functions. The code in figure 1.2 demonstrates how the preprocessor directives in figure 1.1 could be rewritten in C/C++. This is the sort of migration that the method described herein strives to accomplish.

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++" [18] Stroustrup states that use of the C preprocessor should be avoided and details the introduced C++ features to help do just that. Refactoring C or C++ is also made all the more difficult when macros are used [7]. Furthermore the task of migration for C or C++ code, is greatly hampered by the inclusion of macros [15].

```
//warn of any dangerous monsters in vicinity
#define mon_warning(mon) (Warning && !(mon)->mpeaceful && \
        (distu((mon)->mx, (mon)->my) < 100) && \
        (((int) ((mon)->m_lev / 4)) >= flags.warnlevel))

#define WEAPON_SYM ')'
#define ARMOR_SYM '['
#define RING_SYM '='
#define AMULET_SYM '"'

//These attributes are used in a case label
#define A_CHAOTIC (-1)
#define A_NEUTRAL 0
#define A_LAWFUL 1

struct edog {
    int apport; /* amount of training */
    long whistletime; /* last time he whistled */
    long hungrytime; /* will get hungry at this time */
};
#define EDOG(mon) ((struct edog *)&(mon)->mextra[0])
```

Figure 1.1: Example code with preprocessor directives

```
//warn of any dangerous monsters in vicinity
bool mon_warning(most *mon)
{
    return (Warning && !(mon)->mpeaceful &&
        (distu((mon)->mx, (mon)->my) < 100) &&
        (((int) ((mon)->m_lev / 4)) >= flags.warnlevel))
}

const char WEAPON_SYM = ')';
const char ARMOR_SYM = '[';
cosnt char RING_SYM = '=';
cosnt char AMULET_SYM = '"';

//These attributes are used in a case label
enum {
     A_CHAOTIC = (-1),
     A_NEUTRAL = 0,
     A_LAWFUL = 1
};

struct edog {
    int apport; /* amount of training */
    long whistletime; /* last time he whistled */
    long hungrytime; /* will get hungry at this time */
};
edog *EDOG(monst *mon)
{
    return ((struct edog *)&(mon)->mextra[0]);
}
```

Figure 1.2: Example code with preprocessor directives migrated away

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.

## 1.2 Objective

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 the languages, namely CPP. Ideally in doing this, as few changes as possible to the original source should be made. CPP itself will translate the code into straight C/C++, but the implied semantic information that was encoded in the preprocessor directives is lost, not to mention a great deal of readability. Also, it would be useful to preserve the original functionality as closely as possible, in that minor differences in performance or execution branches are acceptable so long as the program runs as it did before.

The approach taken here is to rewrite the original macros using C/C++ language constructs which mimic the macro as closely as possible. A method to perform this task is outlined, and some early results of its usefulness using a test implementation is presented. While the aim is to

handle as many types of macro use as possible, the implemented system can currently only migrate constant macros and parameterless function-like 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.

Though macros are mostly used for simple tasks, like constants and inlined functions, 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.

While the approach to migration is intended to be as straightforward and easy to perform as possible, as always, the devil is in the details. The following steps are taken in the outlined 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

This method is intended to be as straightforward and intuitive as possible. The hardest part with this project was in discovering the more obscure language rules that the migration engine is likely to run into, such as those discussed in section 3.5.

Originally macros were provided in C was for enhancing the language. Many of these enhancements were incorporated into the creation of C++ [18]. As such, the focus of this thesis 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.

## 1.3 Thesis Overview

The organisation of this thesis is as follows:

Chapter 2 discusses background material and related work. A discussion of the C preprocessor is given in detail, covering both its operation and semantics. A brief description of fact extractors follows. The TA file format, used to represent the facts extracted from the fact extractors used in this project, is also described. To manipulate the extracted facts the Grok interpreter is used, which is therein discussed. Finally this chapter details related works.

In chapter 3 the approach taken to migrate C/C++ macros is described. Some of the pitfalls that one encounters when trying to migrate C/C++ macros is also discussed.

Following that, chapter 4 discusses macros. Macro fact extraction is first covered in this chapter, succeeded by a taxonomy of macros.

The details of the implemented system are described in chapter 5. The architecture and interfaces used in the design and implementation are also discussed.

Results of a case study are presented in chapter 6. Using the implemented system, two popular open source applications were analysed in terms of classification of macros and issues encountered during the migration process.

Finally, chapter 7 concludes with a summary of this thesis, and outlines the salient points made. The chapter ends with a discussion of future areas of work.

# Chapter 2

# Background and Related Work

## 2.1 The C/C++ Preprocessor

In addition to each C/C++ compiler there is a macro preprocessor, which we will call CPP. More than just a convenience, CPP originally helped programmers overcome some of the limitations of C. While most of these deficiencies have been addressed in later versions of C and C++ [18], the preprocessor has persisted out of a need for backwards compatibility with older versions of C/C++. Before the C/C++ compiler actually compiles a given source, the preprocessor is run and that output is compiled. CPP has three main operations: inclusion of files, conditional directives, and macro expansion.

### 2.1.1 `#include` Directive

C/C++ relies on the preprocessor to include files within sources. Most commonly this inclusion is done for the purpose of including header files for the standard C/C++ libraries. The C/C++

| /* File: header.h */<br>#ifndef _HH<br>#define _HH<br>{ some stuff }<br>#undef _HH | /* File: file.c */<br>int a = 5;<br>#include “header.h”<br>int b = 6; | /* What the compiler sees */<br>int a = 5;<br>{ some stuff }<br>int b = 6; |
|---|---|---|

Figure 2.1: Simple `#include` directive examples

compiler only compiles a single, self-contained, source file at a time. As such, CPP is required to follow each inclusion directive encountered, starting at the original source file, and build the resulting code to be compiled. To actually include a file the `#include` directive is used. When an `#include` directive is encountered, CPP stops processing the current file, and continues with processing the file given in the directive. When that file has been processed, CPP returns to processing the original file at the spot it left off. This processing is recursive, and a file may include itself or cyclic inclusion paths may occur. To avoid the potential problem of infinite inclusion, included files are commonly encapsulated with guards using preprocessor conditionals. Some simple examples of the `#include` directive are given in figure 2.1.

As there is no comparable C/C++ facility to the `#include` directive, no migration techniques are proposed for it.

### 2.1.2 Conditional Directives

Conditional directives in CPP are used to include or exclude code for compilation. We find this most useful in the case of configuration management. For example, should we be compiling our source under two different operating systems, we may find that some library functions have a slightly different syntax. In this situation we are not able to use conditionals in C/C++ since both versions of the functions can not be simultaneously compiled. Rather than duplicating the source

and making the changes in each, we can use a preprocessor conditional directive to choose which flavour of call to use. Alternatively, say our code contained a lot of debugging methods which were overly time consuming. If we wished to remove them in the final release we could wrap those calls in CPP conditionals and exclude them during the final build. Another useful, though less pervasive, use of CPP conditionals is when we wish to comment out a piece of code that already contains embedded comments. Since we can not nest C-style comments, we can use CPP when this need arises.

The syntax of the conditional directives is similar to C/C++ conditionals. Each CPP conditional starts with one of `#if`, `#ifdef`, or `#ifndef`. `#ifdef` and `#ifndef` are aliases for "`#if defined(...)`" and "`#if !defined(...)`", respectively. At the end of a CPP conditional block the directives `#endif` or `#else` are used if we want to either terminate the conditional block or add a block for the negation of the condition, respectively. If one wants to nest conditionals, the directive `#elsif` can be used as a substitute for `#else` then `#if`.

Each conditional is judged on a C-like expression. This expression may contain integer arithmetic, bit-wise/logical, and equality operations. In addition to integers, literals may be used in the arithmetic expression. If such a literal is actually a previously defined macro, then the literal is replaced with its expansion. Otherwise the literal is evaluated as 0. There is also the `defined()` operator which evaluates to 1 if the literal parameter given to it is a declared macro, and 0 otherwise.

To illustrate CPP conditions, some examples are provided in figure 2.2.

In terms of migrating conditional directives, one possible approach would be to replace the directive with a regular C/C++ `if` statement. Even if the directive is an `#ifdef`, `#ifndef`, or uses `defined()` it could still be replaced with a regular C/C++ `if` statement. In this later case

| | | | |
|---|---|---|---|
| `#if 4+5 == 9`<br>`...`<br>`#else`<br>`...`<br>`#endif` | `#ifndef _HH`<br>`#define _HH`<br>`{ stuff }`<br>`#endif` | `#if 0`<br>`{ code with`<br>`comments }`<br>`#endif` | `#if defined(CPU1)`<br>`call(a);`<br>`#else`<br>`call(a,b);`<br>`#endif` |

Figure 2.2: CPP Conditional Examples

| | | |
|---|---|---|
| `if (4+5 == 9) {`<br>`...`<br>`} else {`<br>`...`<br>`}` | `if (_HH == 0) {`<br>`{ stuff }`<br>`}` | `if (0) {`<br>`{ code with`<br>`comments }`<br>`}` |

Figure 2.3: Migrated CPP Conditional Examples

the macro literal being evaluated could itself be migrated as a flag, and the migrated C/C++ `if` statement would then evaluate that flag.

In terms of the examples given in figure 2.2, only the first and third examples could be migrated in this way. If the second example did not contain the line "`#define _HH`", we could also use this technique to migrate it. With this change in mind, the first three examples in figure 2.2 could be migrated as shown in figure 2.3, provided the result is valid C++.

An alternate approach would be to decide upon a static configuration and simply include or exclude the evaluated conditional blocks, exactly as the preprocessor would. This approach would be necessary to handle the fourth example in figure 2.2.

### 2.1.3 Macro Expansion

At its simplest, declared CPP macros are used as aliases for other expressions. Any expression can be declared as a macro expansion, be it valid C code or not. For example, the following are valid

macro declarations:

```
#define foo 5
#define fnord else } ; ...
```

Beyond simple aliases or abbreviations, macros may be declared to take any number of parameters as well. Whatever is passed as a macro parameter is used verbatim in the macro's expansion. To clarify, consider the following snippet of code:

```
#define mac_func(a) 1 a 2
mac_func(blah)
```

When CPP processes this code the expansion of the macro given on the second line is replaced with "`1 blah 2`". A common use of parameterized macros is as a form of inline functions.

As hinted at in the above examples, macros are declared using the `#define` directive. For a parameterized macro, a set of brackets with a comma delimited list of arguments is appended to the macro name. Following the name and optional argument list is the macro expansion. Once a macro is declared any further instances of the macro's name, outside of CPP directives, will be replaced with the macro's expansion.

Whenever a macro is expanded, the CPP will recursively replace any instances of literals that are the name of declared macros with the respective macro's expansion. If we had the following declarations:

```
#define a 1 2
#define b a 3
```

then any expansion of `b` would result in "`1 2 3`". We should note however, that the description of macro expansion so far should lead one to conclude that the CPP could be used as a lambda

calculus interpreter. This would be the case if it was not for the feature that once a macro has been expanded, it can no longer be expanded within the same instance. For example, say we made the declarations:

```
#define a 1 2 b
#define b a 3
```

If the CPP were to expand out `a`, we would get "`1 2 a 3`" and not an infinite string of "`1 2`".

Every declared macro remains in scope until either the CPP finishes processing or explicitly brought out of scope using the `#undef` directive. As such, the expansion of a macro may differ between two instances. This could happen if any macros are used in the original expansion, and those macros being used are either removed from scope or redeclared. To clarify, observe the following example:

```
int a = 10;
#define a 1 + 2
#define b a + 3
int z = b;
#undef a
int y = b;
```

The resulting code will be processed as:

```
int a = 10;
int z = 1 + 2 + 3;
int y = a + 3;
```

In addition to user defined macros, there are a number of standard predefined macros. For example, the macro `__LINE__` expands out to the current line, whereas the macro `__STDC__` expands to 1 if the C/C++ compiler is ANSI standard C compliant.

The possibilities when using macro expansion are enormous. However, it is usually the case that macros are used to represent a constant value or brief expression [5], like the second and final examples in this subsection. A sample of these sorts of macros and their possible migrations is given in section 1.1 in, figures 1.1 and 1.2.

### 2.1.4 Other CPP Features

Beyond the above three types of operations, the CPP can also perform "stringification" and string concatenation. Stringification involves taking a macro parameter and replacing it with a quoted string version of the given value. In the macro expansion any argument to undergo stringification is prefixed with a `#`. To clarify, say we make the following declaration:

```
#define DBG(msg) printf("Error:  " #msg "\n");
DBG(x == 2)
```

The CPP will then produce the following:

```
printf("Error:  " "x == 2" "\n");
```

The string concatenation operation takes two literals separated by "`##`" and creates a new literal by concatenating them together. In addition to literals two operators which when placed together form a valid operator, such as "+" and "=" or "`!`" and "=" may be concatenated. As an example, consider the following lines:

```
#define decl(typ) typ typ ## _type;
decl(int)
```

The resulting processed code would be:

```
int int_type;
```

These two operations, stringification and string concatenation have no obvious parallel in C/C++. As such no migration techniques are proposed for them in this thesis.

Of interest with CPP is that every directive must be declared on a single line. Consequently, the expansion of a macro is in place and does not affect the compiler's line number tracking. We can certainly declare a multi-line macro, but only if we use line continuation markers, essentially declaring it on a single line. The following example illustrates that given the definition:

```
#define func_mac(a, b) a + 5 \
        * 7 + b \
        + b * a
```

the preprocessor will expand every instance of `func_mac` on a single line as:

```
a + 5 * 7 + b + b * a
```

## 2.2 Fact Extractors

Source code can be seen or represented in a number of different ways. The most obvious is the source itself. The information the source provides us comes in the form of discrete tokens of words, numbers, operators, etc. When compiling the source however, the compiler will produce other representations of the code.

Given the source, the compiler will use the preprocessor to generate a partial representation of the preprocessor directives as it creates the code that will actually be compiled. The term "partial representation" is used because not all information about the preprocessor's operation is stored. For example, the preprocessor does not keep track of each expansion occurrence after said expansion has completed. After the preprocessor has created the code that the compiler will compile, the compiler will create an abstract syntax tree representing the input code. From this abstract syntax tree, assembly code is generated and then assembled into the final binary object.

What we see is that the code goes through many representations, from source to abstract syntax tree to binary object. Each of these representations, while presenting the same basic information, is different from the others.

By creating a convenient to use database of the details of a representation allows us to more easily explore properties of the source code. For example, say we wanted to know what functions made use of global variable X. We could look through the source for every instance of the word "X", and then eliminate all the local variables that overshadow the global variable. After a bit of work we would find our answer. The compiler however has already done this. Its abstract syntax tree representation has to know for each variable named X, which X it is referring to. It would be much easier to query the abstract syntax tree, than parse and partially compile the code ourselves to answer our question.

Because the compiler creates these representations for us, it does not make sense for us to duplicate that effort when examining the code. Rather we should extract the relevant information from each representation and place it in a database. These extracted facts will be represented as a directed tree, or perhaps even a directed graph. Each node will have a type and associated properties. Edges between nodes will represent relations between them. For example, a function

```
$INSTANCE @1 cFunction
$INSTANCE @2 cBlock
$INSTANCE @3 cForLoop
contain @1 @2
contain @2 @3
```

Figure 2.4: Example facts of a `for` loop within a function in TA format

that contains a `for` loop will have a representation including a node for the function, a node for the function's block, a node for the `for` loop, and a containment relation between them, like in figure 2.4. A description of the fact format, TA, is given in section 2.3.

As we extract more out from the abstract syntax tree, we learn about where functions and variables are referenced throughout the code, among other things.

Facts can be generated from the other representations, even if they are not natively presented as a graph. For example, the original source code can be represented as a bunch of word nodes, strung together with an ordering relation. The operation of the preprocessor can be recorded by creating facts about every declaration and expansion, and each conditional encountered. Examples of literal source code facts and preprocessor operation facts are given in figures 2.5 and 2.6, respectively. As we can see, even the simplest of expressions or operations can generate a large number of facts.

Any program which extracts such information is called a fact extractor. For the system this thesis describes, the C/C++ fact extractor CPPX [10] and our own fact extractor, CPP0-CPPX, was used.

### 2.2.1 CPPX

CPPX is a fact extractor written by the University of Waterloo's Software Architecture Group (SWAG). Based on GCC, CPPX can extract facts from any C or C++ program that GCC can

**Original Line of Code:**

```
int a = 6;
```

**Generated Facts:**

```
FACT TUPLE :
$INSTANCE @8 cToken
$INSTANCE @9 cToken
$INSTANCE @10 cToken
$INSTANCE @11 cToken
$INSTANCE @12 cToken
AndThen @8 @9
AndThen @9 @10
AndThen @10 @11
AndThen @11 @12
FACT ATTRIBUTE :
@8 { type = CPP_NAME value = "696e74" file = "test.cc" line = 1
        sourceColumn = 1 }
@9 { type = CPP_NAME value = "61" file = "test.cc" line = 1
        sourceColumn = 5 }
@10 { type = CPP_EQ value = "3d" file = "test.cc" line = 1
        sourceColumn = 7 }
@11 { type = CPP_NUMBER value = "36" file = "test.cc" line = 1
        sourceColumn = 9 }
@12 { type = CPP_SEMICOLON value = "3b" file = "test.cc" line = 1
        sourceColumn = 10 }
```

Figure 2.5: Example literal source code facts

**Original Code:**

```
#define add5(x) x+5
int a = 6;
int b = add5(a)
```

**Generated Facts:**

```
FACT TUPLE :
$INSTANCE @15 cMacroDecl
$INSTANCE @21 cMacroParameter
$INSTANCE @31 cMacroExpansion
cMacroParameter @21 @15
cMacroExpansion @31 @15
FACT ATTRIBUTE :
@15 { name = "add5" file = "test.cc" line = 3 startLine = 2
        params = 1 }
@21 { name = "x" parameterIndex = 0 }
@31 { file = "test2.cc" line = 5 }
```

Figure 2.6: Example of preprocessor operation facts

compile. Currently only the abstract syntax tree representation facts are extracted.

## 2.3 TA Modelling Language

CPPX by default uses a subset of TA [9] as its fact extraction output. TA, or the Tuple-Attribute language, is a simple language that can be used to represent graphs. There are two main parts to a TA factbase: the tuples and the attributes.

The tuple section is used to declare nodes and relations between them. Each node must first be defined with a type using the "`$INSTANCE`" relation. A function node may be declared like:

```
$INSTANCE @2 cFunction
```

What this means is that the node `@2` is of type `cFunction`. The node name does not need to be prefixed with a "`@`", it is simply something CPPX does. While "`$INSTANCE`" is a special relation whose nodes should be declared before being used in any other relations, it is treated the same as any other relation in the tuple section. In general a relation tuple is declared like:

```
<Relation name> <From node> <To node>
```

If we wanted to represent the inclusion or containment of a `for` block within a function we could use something like:

```
$INSTANCE @2 cFunction
$INSTANCE @3 cBlock
$INSTANCE @4 cFor
contain @2 @3
contain @3 @4
```

```
FACT TUPLE :
$INSTANCE @2 cFunction
$INSTANCE @7 cBlock
$INSTANCE @27 cForLoop
contain @2 @7
contain @7 @27
FACT ATTRIBUTE :
@2 { file = test.cc line = 3 name = f visb = pub extern = true }
@7 { line = 5 }
@27 { line = 4 }
```

Figure 2.7: Complete TA example

In this very simple way we can represent any directed graph, as the "`$INSTANCE`" relation declares the nodes and the other relations represent the edges between them. There are no restrictions on the number of relations that may exist between two nodes. This leads to the understanding that beyond simply representing directed graphs, TA in fact represents directed multigraphs.

The attribute section of a TA factbase is just as simple as the tuple section. Every node with attributes includes a single entry in the attribute section. This entry lists attribute and value pairs in much the same way as an attribute list for an XML element. Continuing with our function example, its respective attributes could look like:

```
@2 { file = test.cc line = 3 name = f visb = pub extern = true }
```

A complete TA factbase might look like the one given in figure 2.7.

### 2.3.1 TA Macro Schema

As CPPX only extracted facts from the compiler's AST, it was necessary to write our own macro fact extractor. Our approach to this was to take the GNU GCC C preprocessor and modify it to

export facts about its operation. Beyond the immediate changes necessary to accomplish this, a schema for the extracted facts needed to be created. We informally describe here the scheme used to describe the macro facts.

### Macro Declarations and Expansions

Every macro declaration creates an instance tuple of type `cMacroDecl`. Associated with this tuple are the attributes `name`, `file`, `line`, `startLine`, and `params` which represent the name of the macro, the file in which it was declared, the line after the end of the macro declaration, the line the macro declaration started on, and the number of parameters the macro takes, respectively. For example, the statement "`#define func(a) 5 + a`" could create the following instance tuple and attributes:

```
$INSTANCE @15 cMacroDecl
@15 { name = "func" file = "test.c" line = 6
        startLine = 5 params = 1 }
```

Each parameter of a macro also creates an instance tuple. In this case an instance tuple of type `cMacroParameter` is created. This tuple has the associated attributes `name` and `parameterIndex` which represent the name of the literal as given in the macro declaration and the ordered index of the parameter, respectively. We also need to associate the parameter with its containing macro. This is done by creating a `cMacroParameter` tuple. Continuing with our example above, we would include the following facts in our TA file:

```
$INSTANCE @21 cMacroParameter
cMacroParameter @21 @15
@21 { name = "a" parameterIndex = 0 }
```

The final set of facts extracted from our macro declaration have to do with the macro expansion. Each literal token of the macro expansion has an associated cMacroDefinitionToken instance tuple created for it in the extracted facts. Along with this tuple are attributes describing the token's lexical type, literal value, file, line, and starting column, all represented by type, value, file, line, and sourceColumn, respectively. To enforce the order of the tokens in the relation tuples, for every token after the first an entry in the AndThen relation is created, forming a linked list of sorts of tokens for a given expansion. Finally, the first token is associated with the declaring macro through the cMacroDefines relation. To complete our example above, the following facts related to the macro expansion would be added:

```
$INSTANCE @16 cMacroDefinitionToken
$INSTANCE @18 cMacroDefinitionToken
$INSTANCE @20 cMacroDefinitionToken
AndThen @16 @18
AndThen @18 @20
cMacroDefines @16 @15
@16 { type = CPP_NUMBER value = "5" file = "test.c"
        line = 5 sourceColumn = 17 }
@18 { type = CPP_PLUS value = "+" file = "test.c"
        line = 5 sourceColumn = 19 }
@20 { type = CPP_MACRO_ARG value = "a" file = "test.c"
        line = 5 sourceColumn = 21 }
```

Every expansion of a macro generates a set of facts as well. A cMacroExpansion instance tuple is created for each expansion with the associated attributes file, line, and sourceColumn,

to indicate where the expansion occurred. The expansion needs to be associated with the original macro and this is done through the `cMacroExpansion` relation. The final additions to our example above when including the expansion facts would be:

```
$INSTANCE @23 cMacroExpansion
cMacroExpansion @23 @15
@23 { file = "test.c" line = 6 sourceColumn = 1}
```

**Preprocessor Conditionals**

Extracted preprocessor conditional facts begin with an instance tuple that declares their type. By this it is meant that `#if`, `#ifdef`, and `#ifndef` are represented by `cMacroIf`, `cMacroIfdef`, and `cMacroIfndef` instance tuples, respectively. Each of these three instance tuples have associated with them `file`, `line`, and `entered` attributes which represent the location of the directive and whether or not the condition was true, and thus whether or not the conditional block was entered. Both the `cMacroIfdef` and `cMacroIfndef` instance tuples also have a `name` attribute. This attribute specifies the literal value of the macro whose declaration is being queried. We consider the following example to clarify the above paragraph:

| **Original Code** | **Extracted Facts** |
|---|---|
| `#ifdef a` | `$INSTANCE @11 cMacroIfdef` |
| `#endif` | `@11 { name = "a" file = "test.c" line = 29` |
| | `entered = false }` |

At the end of each conditional there is an `#endif` statement, which also has facts associated with it. Namely for each `#endif` statement a `cMacroEndif` instance tuple is created with associated `file` and `line` attributes.

Should the conditional be a little more complicated and involve an `#else` or `#elif`, an instance tuple of type `cMacroElse` or `cMacroElif` would be extracted. These instance tuples have the same `file`, `line`, and `entered` attributes as a `cMacroIf` instance tuple.

Since preprocessor conditionals can be nested, the problem of identifying which conditional statements belong to which conditional block becomes an issue. To avoid parsing the code, which the preprocessor and the fact extractor does anyway, the relation `cMacroBlockPart` is exported in the facts. This relation represents a linked list of conditional statements within the same conditional block. To clarify this point, we consider the following example:

| **Original Code** | **Extracted Facts** |
|---|---|
| `#ifdef a` | `$INSTANCE @11 cMacroIfdef` |
| `#else` | `$INSTANCE @14 cMacroElse` |
| `#endif` | `$INSTANCE @17 cMacroEndif` |
| | `cMacroBlockPart @11 @14` |
| | `cMacroBlockPart @14 @17` |
| | `@11 { file = "test.c" line = 39 entered = false }` |
| | `@14 { file = "test.c" line = 40 entered = true }` |
| | `@17 { file = "test.c" line = 41 }` |

The final preprocessor directive to discuss is the `defined()` operator. When a `defined()` operator is encountered the fact extractor will include a `cMacroDefined` instance tuple. This instance tuple has associated with it `file`, `line`, `sourceColumn`, and `arg` attributes, representing the location of the operator and the name of the macro literal that was searched for. When the literal that was searched for was found a new tuple in the `cMacroSubject` relation is added to the facts, indicating which macro the `defined()` operator operated on. If the searched for

literal is not found, a tuple in the `cMacroSubject` relation is not created. To associate the `defined()` operator with the parent `#if` statement, a tuple in the `cMacroBlockPart` relation is added. The following example will illustrate the facts that are created with a `defined()` directive:

| **Original Code** | **Extracted Facts** |
|---|---|
| `#define a` | `$INSTANCE @11 cMacroDecl` |
| `#if defined(a) &&` | `$INSTANCE @25 cMacroIf` |
| `    defined(b)` | `$INSTANCE @26 cMacroDefined` |
| `#endif` | `$INSTANCE @27 cMacroDefined` |
| | `$INSTANCE @30 cMacroEndif` |
| | `cMacroBlockPart @25 @30` |
| | `cMacroBlockPart @25 @26` |
| | `cMacroBlockPart @25 @27` |
| | `cMacroSubject @26 @11` |
| | `@11 { name = "a" file = "test.c" line = 26` |
| | `        startLine = 25 }` |
| | `@25 { file = "test.c" line = 27` |
| | `        entered = false }` |
| | `@26 { file = "test.c" arg = "a" line = 26` |
| | `        sourceColumn = 13 }` |
| | `@27 { file = "test.c" arg = "b" line = 26` |
| | `        sourceColumn = 27 }` |
| | `@30 { file = "test.c" line = 28 }` |

**Other Preprocessor Constructs**

We discuss here the remaining preprocessor constructs, namely `#undef`, `#include`, stringification and string concatenation.

When encountering an `#undef`, the fact extractor first creates a `cMacroUndef` instance tuple, with the attributes `file` and `line`. A tuple in the `cMacroSubject` relation is then created, linking the `#undef` statement with the macro being undefined. If the `#undef` statement did not undefine an in-scope macro, then no tuple is added to the `cMacroSubject` relation. The following example will illustrate the `#undef` statement facts:

| **Original Code** | **Extracted Facts** |
|---|---|
| `#define a` | `$INSTANCE @11 cMacroDecl` |
| `#undef a` | `$INSTANCE @17 cMacroUndef` |
| `#undef b` | `$INSTANCE @18 cMacroUndef` |
| | `cMacroSubject @17 @11` |
| | `@11 { name = "a" file = "test.c" line = 7 startLine = 6 }` |
| | `@17 { file = "test.c" line = 7}` |
| | `@18 { file = "test.c" line = 8}` |

For every file referenced in the fact attributes the preprocessor fact extractor will create a `cFile` instance tuple. This instance tuple contains a single attribute, `file`, which gives the name of the file. While a reasonable convention, this instance tuple makes more sense in light of the facts extracted from a `#include` directive. When such a directive is encountered, the fact extractor will first create a `cInclude` instance tuple, with `file` and `line` attributes. To indicate which file is being included, a tuple in the `cIncludes` relation is created that links the respective

`cInclude` node with its corresponding `cFile` node. Finally, a tuple in the `contain` relation is created from the file node in which the `#include` directive is found, to the `cInclude` node. To clarify, let us consider the following facts extracted from file `test.c`:

| **Original Code** | **Extracted Facts** |
|---|---|
| `#include “b.h”` | `$INSTANCE @11 cInclude` |
| | `$INSTANCE @18 cFile` |
| | `$INSTANCE @22 cFile` |
| | `contain @18 @11` |
| | `cIncludes @11 @22` |
| | `@11 { file = "test.c" line = 7 }` |
| | `@18 { file = "test.c" }` |
| | `@22 { file = "b.h" }` |

No progress was made towards the migration of stringification or string concatenation preprocessor operators. Because of this, no steps have been taken in formulating extracted facts for these operations.

## 2.4 Grok

The most common means of manipulating TA databases is with a language called Grok [8]. Grok is akin to a restricted form of SQL in that it defines a set of commands for manipulating binary relations and sets. A TA factbase is represented as a set of binary relations. Within each TA factbase the tuple section is reflected in a straightforward manner. The tuple relation name, becomes the relation name under Grok with the two parameters thusly forming part of the relation. For example,

the TA line

```
$INSTANCE @2 cFunction
```

would add the tuple `(@2, cFunction)` to the "`$INSTANCE`" relation in Grok.

TA attributes are also represented as binary relations. Every attribute in an attribute list is represented by Grok as a relation. The relation's entities are then composed of the node name that the attribute is associated with and the attribute value. If we had the following attribute in a TA factbase,

```
@2 { file = test.cc line = 3 name = f }
```

we would represent this in Grok by adding the tuples `(@2, test.cc)`, `(@2, 3)`, `(@2, f)` to the `file`, `line`, and `name` relations, respectively.

To further clarify, say we had the TA factbase from figure 2.7. If loaded into a Grok interpreter, the resulting relations would be:

| **Relation** | **Tuples** |
|---|---|
| `$INSTANCE` | `(@2 cFunction), (@7 cBlock), (@27 cForLoop)` |
| `contain` | `(@2 @7), (@7 @27)` |
| `@_file` | `(@2 test.cc)` |
| `@_line` | `(@2 3), (@7 5), (@27 4)` |
| `@_name` | `(@2 f)` |
| `@_visb` | `(@2 pub)` |
| `@_extern` | `(@2 true)` |

We note that the Grok interpreter appends the prefix "@_" to the name of the attribute relations.

As well as relations, Grok defines commands for manipulating unordered sets and some simple constructs that allow it to be used as a scripting language.

### 2.4.1 Language

The Grok language itself consists of very concise operators for manipulating relations, sets, strings, and numbers, plus some simple control flow constructs. For example, the following script will take as it's first argument a TA factbase's file name whose facts were extracted from a source file. It's assumed this source file contained a function whose body contained a single entity. The name of this function is given to the script as it's second argument. Once it has this information, the script will determine the type of the entity within the named function:

```
addta $1
functionNode := @_name .  {$2}
cBlock := (rng ((id functionNode) o contain))
literalNode := (rng ((id cBlock) o contain))
builtinNode := (rng ((id literalNode) o cInstance))
type := (inv @_name) .  builtinNode
if type == EMPTYSET then
    type := (inv @_name) .  (rng ((id builtinNode) o cInstance))
    addsuffix type "*"
end if
```

This example demonstrates some simple set and relation operators, such as the relational composition operator, "o", and set projection operator, ".", as well as the "rng" operator which returns the range of a relation. The specific details of the script's function aren't important to understanding this example of the Grok language.

As we can see, the language has a straightforward infix notation. Operators such as "+", "-", and "^" are used to represent operations like union, difference, and intersection, respectively, for both sets and relations. As well as simple set operations Grok allows us to do things like create relations through the cross product of two sets or take the transitive closure of a relation.

A full description of the Grok language, including tutorials, examples, and language specification can be found in the introductory paper written by Holt [8].

### 2.4.2 Embedded Grok Interpreter

The Grok interpreter is normally used as a stand-alone application, appropriately called "grok". This interpreter is most commonly used to run scripts that are part of SWAG's Software Architecture Toolkit. An interactive command-line interpreter was also a feature of grok.

As Grok is written in Turing, and then transcoded into C by the Turing compiler, an embedded version of grok was created from the "hijacked" C source. Rather than interpreting commands from the console, grok is tricked into accepting a command directly via a method call. Hooks were added to retrieve relation and set contents. In this way we were able to use grok directly within our system, rather than indirectly through scripts. We use TA for the databases in our system, necessitating the creation of an embedded version of grok.

## 2.5 Related Work

In conducting this research, we were inspired by the work of Ernst, Badros and Notkin [5, 2]. 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 [2]. Along with other applications, the tool was applied to replace macro definitions for constant values with equivalent C declarations [1]. 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 thesis, 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 mainly focus on constant macros in this thesis, 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 [4, 12, 13]. The problem of tracking substitutions through the preprocessor is examined by Kullbach and Riediger [12]. Their *folding* method allows a user to visualize the actions of the preprocessor on a particular construct. Cox and Clarke [4] 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. [14] 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 [17], source code should be re-written or transformed to eliminate these constructs, but in some cases this transformation may be impossible. Somé and Lethbridge [16] discuss many of the problems associated with conditional compilation and describe a parsing method for efficiently processing conditionally excluded code. Others [11, 3] have applied symbolic execution and partial evaluation techniques to analyze conditional constructs.

# Chapter 3

# Approach to Migrating Macros

## 3.1 Fact Extraction

As outlined in section 1.2, we take a straightforward approach to the migration of macros:

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

The first step is to extract the facts about the system. For each compiled source we use our custom preprocessor fact extractor, CPP0-CPPX, to extract the macro and lexer token facts and CPPX to extract the ASG facts. By lexer token facts we mean the same thing as literal source code facts,

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

Figure 3.1: Example of indirect expansion of macro A from direct expansion of macro B

such as those shown in figure 2.5. With these three separate fact files we ensure that each node has a unique name, and then merge the three factbases together. From this new factbase we remove any of the facts that will not be needed; the standard C include header facts in particular. From here we merge the facts into the overall project factbase. Finally, duplicate entries in the factbase 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 the `#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 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 3.1.

In the simplest case, macros which do not depend on any others may be migrated without paying attention to the order in which they are processed. However, we may come across the situation in which for a particular macro to be migrated correctly, every macro that it depends on must also be migrated along with it. In this case we must ensure that these groups of macros are migrated together. Depending on how one handles `#if` statements, one could deem all macros in an `#if` statement condition as being dependent on each other, as illustrated in figure 3.2.

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

Figure 3.2: UNIX, SMALL, and CONFIGURABLE can be seen as being dependent on one another

| Code | Classifications for A | Resolution for A |
|---|---|---|
| `#define A 5`<br>`#ifdef A`<br>`#endif` | 1) Constant Value<br>2) Configuration Setting | 1) Constant Value |

Figure 3.3: Example of classification resolution

## 3.2 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 described in section 4.2. The macro being processed is matched against 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 reflect the classification that mirrors as closely as possible the intent the original coder had when creating the macro (figure 3.3).

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 of the previous pass has been completed. So far

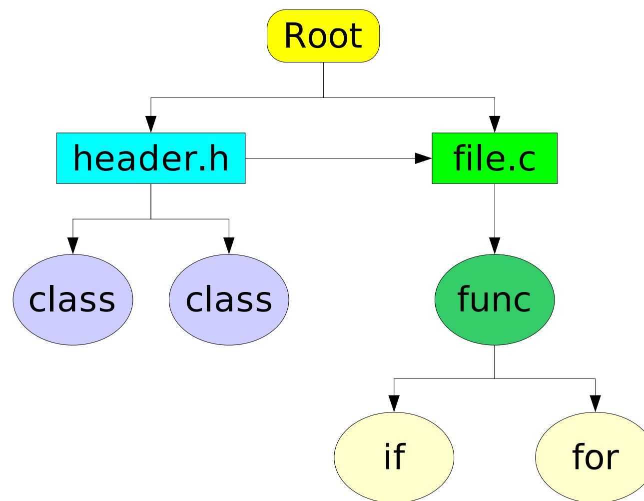


Figure 3.4: Block dependency graph example

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 a `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 3.4.

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

## 3.3 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 is added to the placement method just given. Let us assume that the block in which the macro is declared has a path in the block graph to the discovered least common ancestor block. This added rule states that 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 it is 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 the migrated declaration should remain there if possible (figure 3.5).

Though we know the block in which 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 3.6. We can not declare a variable within a switch statement that has multiple case labels, unless the particular case label contains a proper C block,

**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 3.5: Examples of overlapping scope and original declaration as ancestor to LCA of uses

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 one final task must be performed, in that we must check for any name conflicts which may now arise. As we have possibly moved the declaration to a new location in the source, we are now at risk of causing a name conflict with other entities in the code. To resolve any such

**Legal**

```
switch(something) {
    case 4:
    {
        int macro = 7;
    }
    case 5:
        return;
}
```

**Illegal**

```
switch(something) {
    case 4:
        int macro = 7;
    case 5:
        return;
}
```

Figure 3.6: Switch declaration examples

```
<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 3.7: Example transformation file

conflicts we must decide whether to rename either the migrated macro, or that which it conflicts with.

## 3.4 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 3.7. 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.

## 3.5 Pitfalls in Migration

The approach we have described generally works. 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. While not an exhaustive list of issues, we discuss a few which were immediately encountered in the course of our case study in chapter 6.

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

In this case one must ultimately 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.

Another interesting problem is when we discover that a macro simply can not be migrated. For

**Constant String Concatenation**

```
#define mac_str "constStr"
printf("a" mac_str "b");
```

**Constant String Initializer**

```
#define mac_str "constStr"
char str[] = "a" mac_str "b";
```

Figure 3.8: Un-realizable migration examples

example, if we have a constant string that is used in a constant string concatenation or as a constant initializer (figure 3.8). One could argue that in some, or even most, cases these types of macro instances could be cleverly rewritten to allow migration.

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 face the issue of comments. While beyond the scope of this thesis, if a migrated macro is placed in a different location, then any comments associated with it must also be similarly moved and placed.

When it comes to the nuances of the language, we must accept that there are certain instances of macro use in which we simply can not do anything about them without resorting to significant code re-engineering, which we are trying to avoid. Similarly we may be constrained in how we can handle undefined and/or compiler dependent functionality, such as the comparison of pointers between equivalent string constants. We briefly discuss how we could handle the differences in language dialects and compilers in section 5.3.

### 3.5.2 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. However, consider the case if the macro within the expansion is not in scope, relative to the encapsulating macro, some of the times it is used. In this situation 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 encounter this problem of the macro having different semantics depending on where it is used. In some cases we could duplicate the original macro based on its different instances of use, but this leads to readability/maintainability problems and naming issues.

# Chapter 4

# Macro Fact Extraction and Classification

## 4.1 Macro Fact Extraction

Normally C/C++ fact extractors provide a representation of the code which closely resembles the abstract syntax graph (ASG) that 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.

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

Figure 4.1: Simple code fragment with preprocessor directives

### 4.1.1 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 [10] 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 4.1 provides a simple example, which results in the set of facts shown in figure 4.2. A more detailed description of the fact schema used for CPP0-CPPX is given in section 2.3.1.

Like CPPX, we represent our macro facts in the TA language [9], which was briefly described in section 2.3. 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.

```
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 4.2: Example macro facts in TA format

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

Figure 4.3: Simple code fragment with multi-line macro declaration

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 4.2 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 part of the conditional was actually included in the output code. As CPP processed directives within each part, regardless 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" part 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 4.3 with corresponding facts shown in figure 4.4.

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

```
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 4.4: Example multi-line macro facts in TA format

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. It was 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 and the normal operation of most Unix C compilers.

### 4.1.2 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 4.5 and 4.6. To be consistent with CPPX and CPP0-CPPX we again used TA as the output.

```
a = 16;
```

Figure 4.5: 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 4.6: Example token facts in TA 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.

## 4.2 Macro Classifications

We have classified macro use into almost two dozen types, similar to those found in Ernst et al. [5]. Our taxonomy is inspired by the styles of macro use that were encountered in the systems used for the case study in chapter 6. Presented below is a brief description of each of these classifications:

### 4.2.1 Unparameterized Macros

#### Simple Constant

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*/
```

**Enumerated Constant**

These macros could be classified as either Simple Constants or Constant Expressions. What sets these Enumerated Constants apart is their use in a `case` label in a `switch` statement. By being used in a `case` label, we assume the intent was to use the value as an enumeration.

```
#define VALUE 5
switch(Length) {
    case VALUE:
        break;
}
```

**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 these variables are never overshadowed by others in any of the macro expansions.

```
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 do not support certain keywords.

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

**Code Snippet**

These macros do not expand out to well formed C/C++ expressions, but instead their expansions are snippets of code.

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

### 4.2.2 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 do not expand out to well formed C/C++ expressions, but instead their expansions are snippets of code.

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

#### Projection 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
```

### 4.2.3 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*/
```

### 4.2.4 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);
```

## 4.3 Macro Migrations

While section 4.2 and chapter 5 provide a macro taxonomy and architectural details of the implementation, respectively, more detail of the actual macro migration is required. The migrations performed by the implementation, along with thoughts about those not implemented are described in the following subsections.

### 4.3.1 Implemented Migrations

For macros classified as `Empty Declaration` and `Keyword Alias`, the migrations performed are to simply do what the preprocessor would have done. In this case references to the macro are removed, or substituted with its expansion, respectively.

The two constant types, `Simple Constants` and `Constant Expressions`, are migrated the same way. Once the type of the expansion is inferred, the declaration is replaced with a typed assignment of the macro literal to its expansion. In clearer terms, say we had the directives:

```
#define val 5
#define val2 val + 9
```

They would be migrated to the statements:

```
const int val = 5;
const int val2 = val + 9;
```

In the case of an `Enumerated Constant`, the migration is slightly more complex. Say we again had the directives:

```
#define val 5
#define val2 val + 9
```

and for the sake of argument, let us assume they were used in case labels within the same switch statement. The following statement would be the result of their migration:

```
enum {
    val = 5,
    val2 = val + 9
};
```

Alternatively, if the two expressions were used in case labels in different switch statements, their resulting migration would be:

```
//Somewhere in code
enum {
    val = 5
};
//Possibly somewhere else in code
enum {
    val2 = val + 9
};
```

Macros classified as `Parameterless Functions` are rewritten as inlined functions. The type of the original macro's expansion is used as the new function's return type, and a function returning that expansion is created. Say we had the following expressions:

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

We could migrate the two `Parameterless Functions` as:

```
inline int TwiceScore()
{
    return (Score * 2);
}
inline void Silly()
{
    { int x = 5; x = x + 2; }
}
```

As well, all the previous expansion references of `TwiceScore` and `Silly` must now be appended with `()`.

### 4.3.2 Possible Future Migrations

#### Declarative Directives

For macros classified as `Inlined Functions`, the migration is similar to that of `Parameterless Functions`. The only difference now is that there are parameters in the directives. To illustrate, consider the following directives:

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

Assuming the types of all the parameters is `int`, the directives could be migrated as:

```
inline int TwiceScore(int Score)
{
    return ((Score) * 2);
}
inline int SillySum(int X, int Y)
{
    return (X+Y+1);
}
```

Complications arise when a macro looks like a `Parameterless Function` or `Inlined Function` type but the variables used in its expansion either refer to different variables, or variables declared in a scope in which the macro can not be migrated into. To clarify, we consider the directive:

```
#define AlterScore(X) Score = Score << X
```

The complications arise if `AlterScore` is called from `FunctionA` and `FunctionB`, and both those functions declare a local variable called `Score`. Alternatively, if `AlterScore` is declared and only used within `FunctionA`, we still have a problem with its reference to the local variable `Score`. In this later case, and if we're targeting C++, then we can't migrate the macro to a position within `FunctionA`, as C++ prevents us from declaring functions within functions.

Both these situations are resolved in the same way, and are the conditions for when a macro should be classified as either a `Parameterless Function With Variable Use` or `Inlined Function With Variable Use`. Instead of referencing the conflicting variables directly, they must now be passed in by reference. In the case of a macro classified as a `Parameterless`

`Function With Variable Use`, the result would be the same but without any original parameters. Assuming the type used for `Score` and `X` is always `int`, the above example declaration for `AlterScore` could be migrated to:

```
inline int AlterScore(int X, int &Score)
{
    return Score = Score << X;
}
```

Each call to `AlterScore` must then be updated to pass in the `Score` variable. We have also been making the implicit assumption that the parameters passed in for `Inlined Functions`, `Inlined Functions with Variable Use`, and `Parameterless Functions With Variable Use` macro types are always of the same type. If this is not the case then the current solution, though far from ideal, would be to create enough duplicates of the migrated function to cover the combinations of parameter types. Each duplicate function may need to have a different name, depending on if the variants in parameter lists can be unambiguously distinguished or not.

The final declarative macro style to be discussed is the `Variable Alias` type. As long as the variable being aliased is in scope for all the original macro expansions, and the target language C++, we can migrate the given macro as a reference to the expanded variable. For example, if we had the expressions:

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

we could migrate `HEIGHT` to:

```
int &HEIGHT = CurrentBuildingHeight;
```

However, if we were to target C, or if the aliased variable changed, we would simply migrate the macro to a pointer reference. The migrated reference would be assigned to the pointer of the variable being aliased, and all references to the original expansion occurrences would then need to be dereferenced. This is assuming that in the case of the aliased variable changing, the type of the aliased variables remained the same. If the types of the aliased variables change, then we could create multiple instances of the migrated macro. Each such instance would reflect the different types required, have a different name, and the appropriate expansion instances would be renamed accordingly. Obviously a lot more work is required should the types involved change.

### Conditional Directives

Among the conditional directive classifications, the `Configuration Setting` type is the most difficult to migrate. Unless the code within the conditional directive's parts are valid statements, it is assumed that migrating the conditional would require a significant amount of refactoring and would be suited more to a tool specifically meant for configuration management.

Otherwise, an approach to migrating this type of conditional would be to simply convert it into a normal C conditional, where possible. If the conditional appears outside a C block, then we could simply include all the parts of the conditional, as if the preprocessor had evaluated it to both true and false. Of course this can only work if the code in the respective blocks do not conflict with each other. For example, in the situation of a conditional directive being used to choose a function declaration, we again find that the migration isn't necessarily possible. The difficulties in dealing with code that uses conditional directives are well known, and due to these difficulties no satisfactory migration approach is suggested in this thesis.

Dealing with the other conditional directive classifications is much simpler. `Header Sentinels`

can not be migrated, unless one uses a `#pragma` directive that emulates the same effect. Depending on the intent behind code that is expressly included or excluded using conditional directives, we could simply keep or remove the affected code. For the `Expressly Included Code` type, if the directive appears within a C block, the directive could be migrated into a regular C `if` statement. However, if the intent behind explicitly including or excluding code is to allow an easy means of choosing whether to include or exclude the affected code blocks, then such directives should not be migrated.

### 4.3.3 Current Unknown Migrations

The remaining macro types from section 4.2 that were not discussed in sections 4.3.1 or 4.3.2 are not presented with any suggested migrations in this thesis. However, it is expected that in general `Code Snippet`, `Token Pasting`, and `Literal Expansion` macro types to be impossible to migrate. However, approaches to migrating the remaining types may be possible.

# Chapter 5

# Implemented Architecture and Framework

The implementation of the outlined migration process consists of three main phases. First there is the *Fact Extractor* phase, then the *Migration Engine* phase, and finally the *Source Transformer* phase. Each phase in the migration pipeline is associated with a corresponding architectural component in the high level architecture (figure 5.1).

The *Fact Extractor* analyses the input source and creates a fact database that contains information about the code. Included in this database are the extracted facts from the compiler's abstract syntax tree. As well, facts about the operation of the preprocessor and all the tokenized lexical data from the source code is included.

Most of the analytical work is done in the *Migration Engine*. This component takes the facts created by the *Fact Extractor* and determines all the transformations necessary to migrate each macro.

Once the *Migration Engine* has completed, the *Source Transformer* uses the generated transformations to migrate the original code.

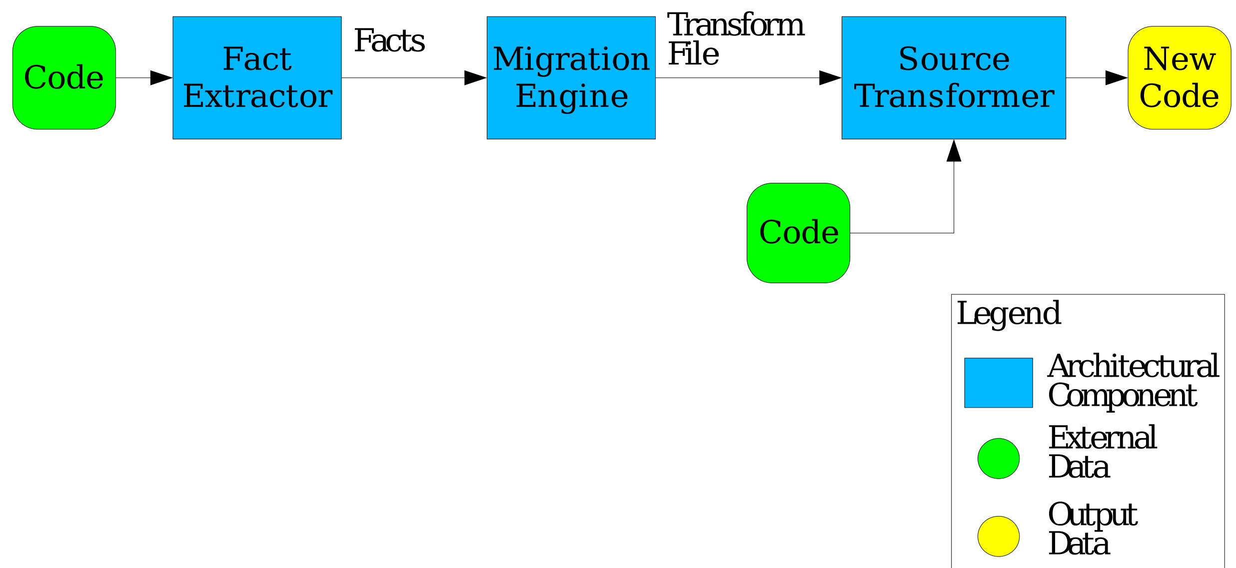


Figure 5.1: High Level Architecture

## 5.1 Document Conventions

For the purposes of this chapter on the implemented architecture and framework, the following conventions will be used:

When discussing an architectural component that corresponds to a component found in a figure, italics will be used in the name of said component.

## 5.2 Fact Extractor

The Fact Extractor, whose high level architecture is shown in 5.2, begins by using CPPX and CPP0-CPPX to extract the C/C++ ASG facts, macro facts, and source code token facts. Each of these three sets of facts are initially represented by three separate TA fact files. Remembering that the fact extraction is repeated for every compiled source file, after each set of facts is generated

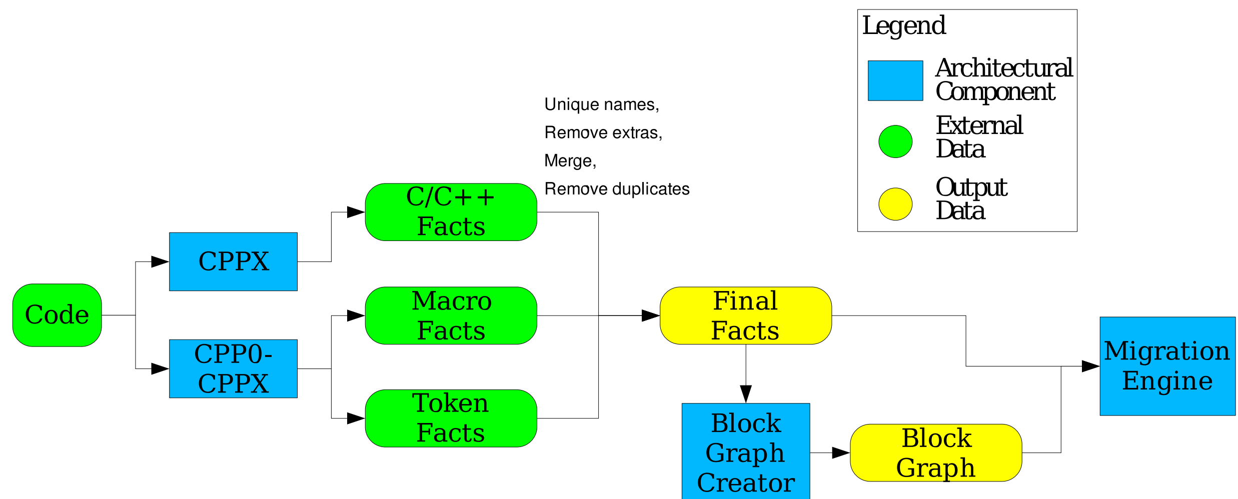


Figure 5.2: Fact Extractor Architecture

additional prep work must be performed.

First, each node in the fact files is provided with a unique name by appending the filename of the original source file and a fact type specifier (original, macro, token) to the name of each node. The fact type specifier has no purpose other than the guarantee the uniqueness of all the node names. Once that is done, the three fact files are merged together. From there all references to files that we are not interested in, such as the standard system headers, are removed. This removal is accomplished by maintaining a list of the source files to include in the final facts, and simply removing references to everything else. We then add these facts to the entire system's factbase, represented by the component *Final Facts* in figure 5.2. In all likelihood duplicate facts will have been added, and so we search for nodes whose type and attributes are the same. These duplicate nodes are then removed and references to the removed nodes are changed to the remaining one.

Most of the manipulation of the fact files is done using Grok [8], with the node duplicate removal being done through a tool written specifically for this purpose.

After the system has gone through all the compiled source files and built the factbase for the system, the process is finished by creating a block dependency graph using the *Block Graph Creator*. This graph is created by using the *Final Facts* to determine where each block lies in the original source. Within the *Final Facts* every pair of fact nodes representing "curly" braces is located. Every code block inferred from these node pairs is then examined to determine the type of that block. By looking at the facts which relate to the block's respective beginning line number, the block type can be determined. This block graph is then stored in a separate file.

An intentional omission from figure 5.2 are components representing the macro dependency graph and its creator. For the current implementation, they were found to be unnecessary. However, as more macro classification types are handled we will need to decide the specific order in which to migrate them. To do this, a graph of all the macro dependencies, as defined in section 3.1, will be created and used by the *Driver* of the *Migration Engine*.

## 5.3 Migration Engine

The heart of the implementation is contained within this component (figure 5.3). This component is responsible for determining the type of each macro and how to transform them.

### 5.3.1 Driver

The *Driver*'s responsibilities begins with using the *Final Facts* and *Block Graph* to decide the order in which to process the macros. If there was a macro dependency graph, that would be used to find small subgraphs of dependent macros. These subgraphs would consist of dependent macros that must be processed simultaneously for the resulting transformations to give valid and compilable

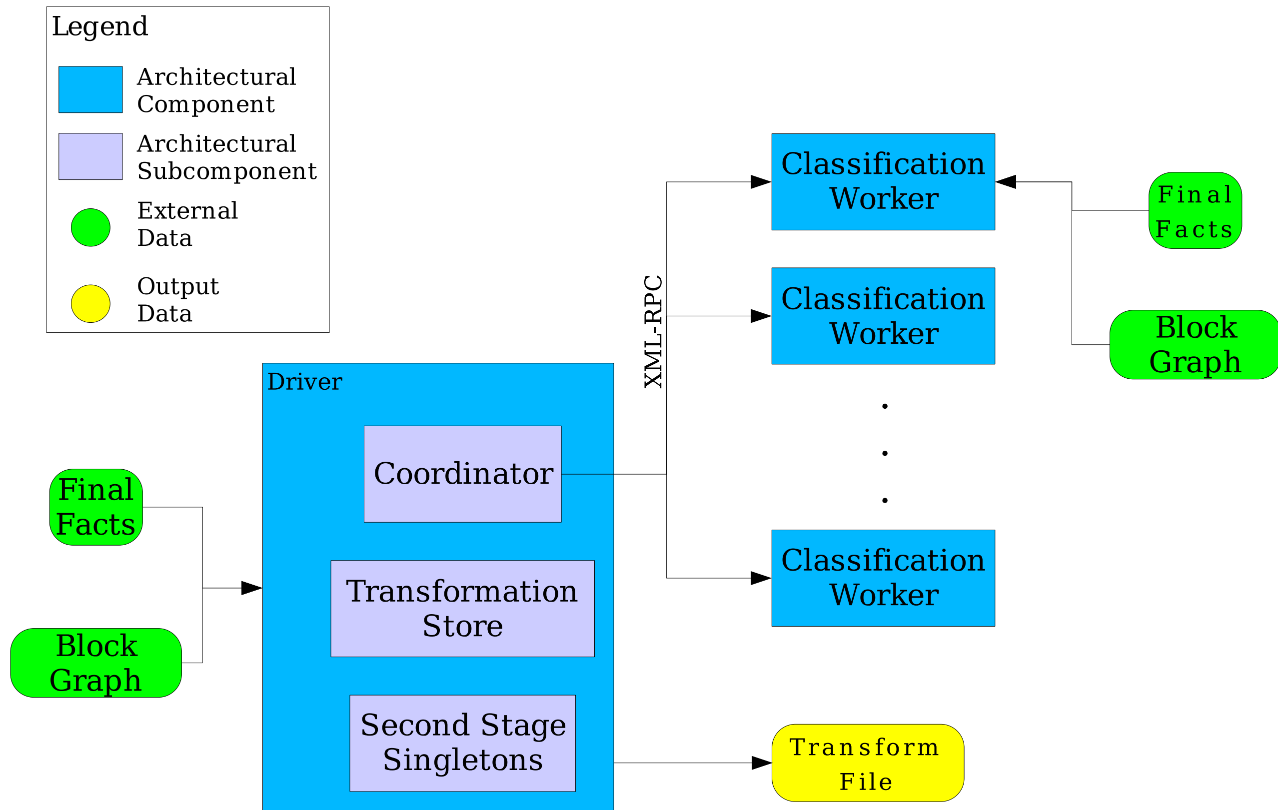


Figure 5.3: Migration Engine Component

results.

Once the macros are divided up into these work allotments, the *Driver* waits for the *Coordinator* to collect the transformations necessary to migrate every macro. After this finishes, the *Driver* allows each *Second Stage Singleton* to perform their necessary processing and transformation generation. Finally, the *Driver* uses the *Transformation Store* to create the *Transform File*. This file is used by the *Source Transformer* to specify the migration steps that it needs to take.

### 5.3.2 Coordinator

For each macro, the classification to apply to it is determined for the *Coordinator*. This is done by giving the macro node name to a *Classification Worker*, and having it determine the best classification type for the given macro. With the macro classification returned to the *Coordinator*, the *Coordinator* gathers any extra facts and then generates the transformations necessary to migrate the original macro. These later two steps are also performed by using a *Classification Worker*.

The *Coordinator* accomplish these tasks by create a 1-1 mapping of threads to *Classification Workers*. In terms of figure 5.7, we call these threads *Coordinator Threads* . Each thread in the *Coordinator* is responsible for processing a single macro at a time. For each macro, the thread requests from the associated *Classification Worker* the macro's classification, fact refinement, and finally the required transformations. Once done, the thread adds the resulting transformations to the *Transformation Store*.

It may also discover from the *Classification Worker* that, in addition to the required transformations, certain second stage operations need to be performed. This information is returned to the thread in the *Coordinator* along with the required transformations. The corresponding *Second Stage Singleton* is then updated with the necessary information for later processing.

Due to the time required for each of the three requests made by every thread in the *Coordinator*, the *Classification Workers* may be distributed to other machines. To this end, each thread in the *Coordinator* uses XML-RPC to communicate with its respective *Classification Worker*.

The choice of XML-RPC was due to the simplicity of the protocol implementations and the availability of libraries.

### 5.3.3 Transformation Store

This component is responsible for collecting all the transformations generated by the *Coordinator*. As each transformation is added it is sorted with the rest, maintaining an ordering based on the file name and affected line number of the transformation.

The *Transformation Store* is also responsible for writing out the *Transform File*.

### 5.3.4 Second Stage Singletons

Certain operations need to be deferred until after the initial macro processing is completed. This component contains singleton structures necessary to collect such information.

For example, when handling simple constant macros that belong in `enum` statements we want to group those macros together in a reasonable fashion, as discussed in section 5.3.4. For this to occur we need to examine the results of first processing each macro, before creating the enumeration transformations.

Since this information is derived from the transformation gathering phase of each thread in the *Coordinator*, we decided to use singletons for the sake of convenience. Once each macro has been processed in the first processing stage and their transformations collected, the *Second Stage Singletons* complete their necessary processing and add the resulting transformations to the

*Transformation Store*.

Arguably the threads in the *Coordinator* and *Classification Workers* should be used to handle the processing cost incurred by each *Second Stage Singleton's* work. However, in the current implementation the processing cost was minimal and thus the distribution of work was found to be unnecessary.

#### Enum Factory

Currently only a single *Second Stage Singleton* has been required; that being the *Enum Factory*. When a macro is used within a case label and will be migrated into an enumeration, the migration is not performed in isolation as with all the other currently implemented classification types. While the migrated macro could be placed into an enumeration with only the one value, it is likely that there are other migrated macros that would logically belong in the same enumeration as the first.

The *Enum Factory* is given the name and value of each such macro and groups them together based on in which switch statements each macro was used. Once the macros have all been processed and the *Second Stage Singletons* begin their processing, the *Enum Factory* creates a list of transformations where macros used in the same switch statements are migrated to the same enumeration.

### 5.3.5 Classification Worker

The *Classification Worker* is a separate process from the *Driver* that communicates with the *Coordinator* via XML-RPC. Each *Classification Worker* is a stateless entity that, given a macro node or *Classification Type*, will perform the desired operation and return the results back to the *Coordinator*. Each *Classification Worker* only services one request at a time. A diagrammatic overview

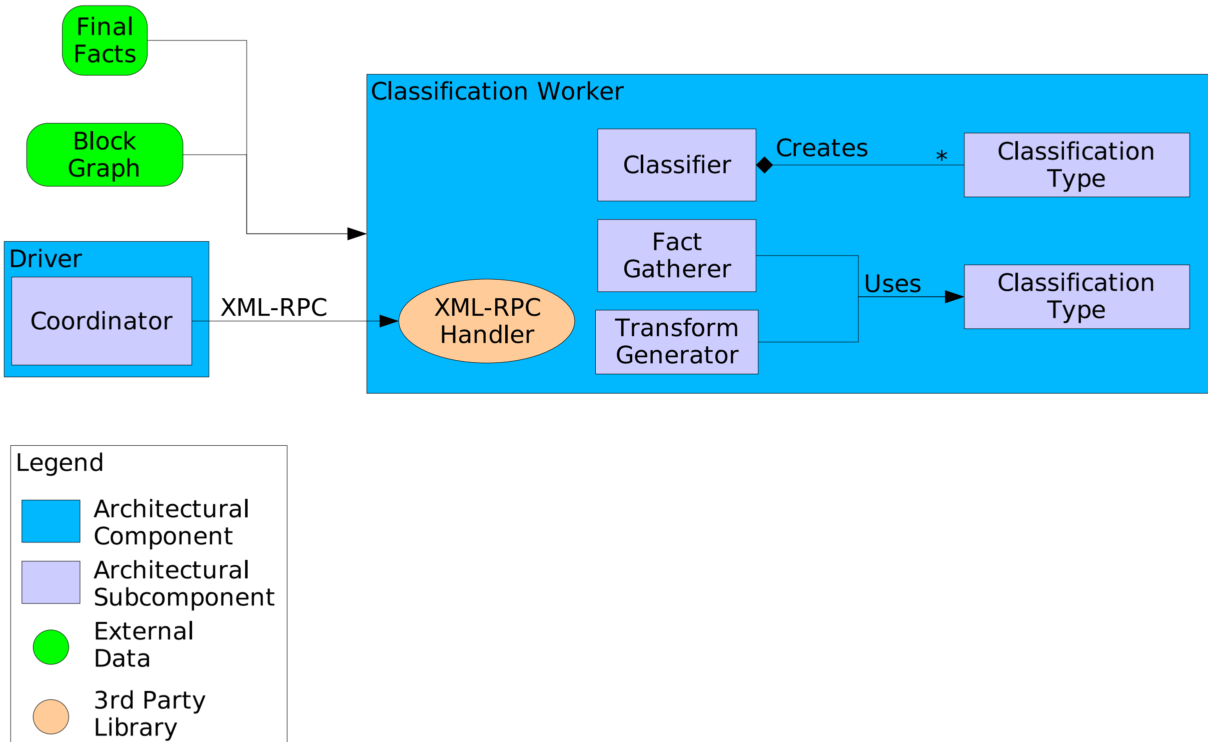


Figure 5.4: Classification Worker Component

of the *Classification Worker* is given in figure 5.4.

There are three RPC method components in this framework: the *Classifier, Fact Gatherer,* and *Transform Generator*.

### Classifier

The *Classifier* is responsible for determining the classification type of a macro, given a macro node. This is done by first creating a generic list of all the *Classification Types*, represented as flyweight objects. Each *Classification Types* object is asked in turn through its *Classification Checker* (figure 5.5), if the macro fits its respective classification criteria. The list of matched criteria is then taken

and a heuristic is applied to choose the “best” classification from the possible choices.

Once we determine the “best fit” classification, a clone of the corresponding flyweight *Classification Type* object is created and returned to the *Coordinator*. We note that in returning the cloned object there will also be some stateful information returned with it, such as the macro value or macro type. The next time the *Classification Worker* is used, this information is reset, though it will of course persist in the clone that was returned to the *Coordinator* for the next request it makes.

One of the intended consequences of using this generic method of managing *Classification Type* 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 the modularity of the *Classification Type* objects allows the flexibility to choose the target environment.

**Fact Gatherer**

Given a *Classification Type* object, the *Fact Gatherer* uses it to collect or infer facts not required for classification or directly necessary to generate transformations. The updated object is returned back to the *Coordinator*.

While not strictly necessary, as this method could be merged into either the *Classifier* or *Transform Generator*, we introduce it as a means to increase the granularity of the work performed by each *Classification Worker* per RPC call.

### Transform Generator

The *Transform Generator* is responsible for generating the transformations necessary to migrate the original macro associated with the given classification type object.

Whatever information that is necessary to migrate the macro, and has not yet been discovered by either the *Classifier* or *Fact Gatherer*, will be done so in this component. For example, simple constants have their expansions recorded and type of variable to create inferred, which is all gathered during the classification stage by the *Classifier*. Determining the declaration line is handled by the *Fact Gatherer.* Other information regarding placement of the migrated macro, such as resolving naming conflicts or placement conflicts, is determined during the transformation stage.

The transformations are presented as a generic list of transformation objects, sorted by file and line. This list is returned to the calling *Coordinator*.

Similar to the *Fact Gatherer*, this component does not do any of the work directly, but instead delegates it to the given *Classification Type*.

### Classification Worker XML-RPC Interface

The XML-RPC interface used by the *Coordinator-Classification Worker* interface is a very simple one and unsurprisingly corresponds to the three main subcomponents of the *Classification Worker*. For the sake of being thorough, and to provide part of a concrete framework API, we present this XML-RPC interface:

ClassificationType **ClassifyMacroNode** (macroNode: **string**)

*macroNode*: The node in the fact base corresponding to the macro declaration of the macro we wish to determine the classification of.

**Returns**: A *Classification Type* object of the determined classification type. This object will contain some facts specific to the macro, depending on the classification type and the information required to classify it.

ClassificationType **GatherMacroFacts** (macroData: ClassificationType)

*macroData*: The *ClassificationType* for the macro we are currently processing.

**Returns**: A *Classification Type* object with the updated datum stored in it.

list<Transformations> **GenerateTransforms** (macroData: ClassificationType)

*macroData*: The *ClassificationType* for the macro we are currently processing.

**Returns**: A list of *Transformation* objects representing the required transformations for migrating the macro associated with the input parameter.

### 5.3.6 Classification Type

For each classification type (section 4.2) that the system handles, a corresponding *Classification Type* class will exist. Each of these classes derives from a *Classification Type* interface class, which is described here and in figure 5.5. To avoid clutter in figure 5.5 a symbol was used to represent the $m \times n$ use of the *Conflict Resolver* subcomponents by the *Interface* subcomponents. By this it is meant that each *Interface* subcomponent makes use of every *Conflict Resolver* subcomponent.

There are three main parts to the *Classification Type* class: the *Interface*, *Conflict Resolvers*, and the *Classification Facts*.

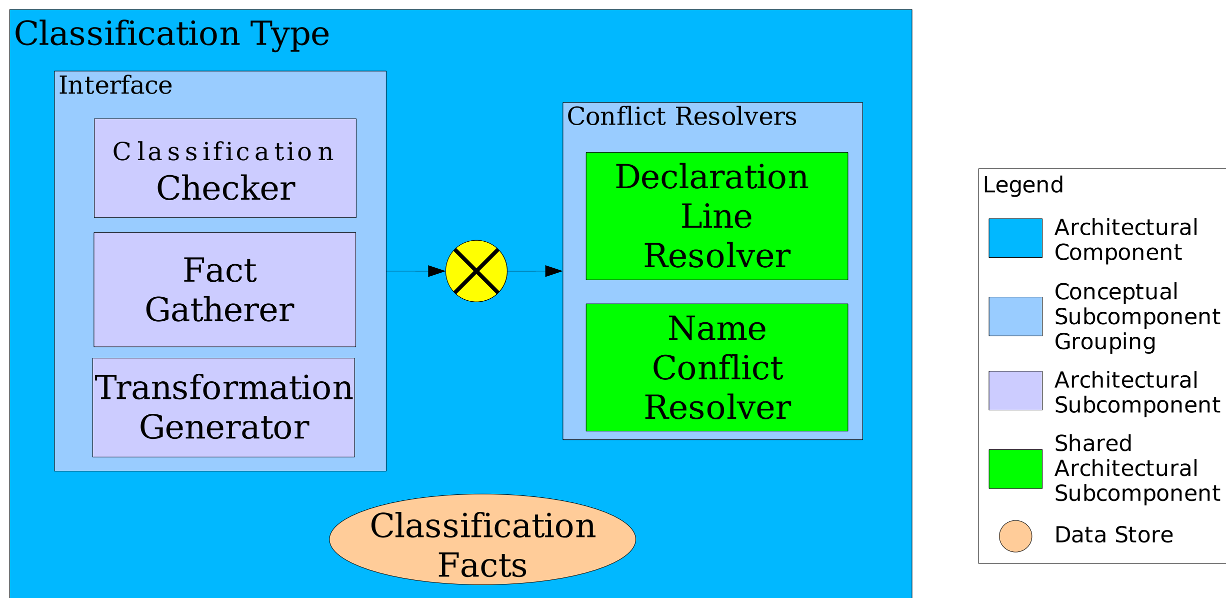


Figure 5.5: Classification Type Component

**Interface**

As alluded to in section 5.3.5 most of the work done by the *Classification Worker* is through using the *Classification Type* class's *Interface*. Corresponding with the three RPC requests for the *Classification Worker*, the *Interface* contains a *Classification Checker, Fact Gatherer,* and *Transformation Generator*.

The *Classification Checker* of a *Classification Type* class is responsible for deciding if a given macro is of the classification type associated with that class. Certain classification types can be ruled out if the *Classification Checker* is aware of the previously matched classifications for the macro by other *Classification Type* classes. While strictly not necessary, the list of previously matched classifications is used to avoid duplicating the effort of prior classification attempts. For example, if a macro is a Simple Constant (section 4.2) but is also used in a case label, then we

ultimately want the macro to be classified as an Enumerated Constant type. The *Classification Checker* for the Enumerated Constant type simply looks through the given matched list, and if it determines that the macro was matched as a Simple Constant and used in a switch statement, it then knows that the macro should be classified as an Enumerated Constant. It is up to the *Classifier* in the *Classification Worker* to ultimately favour the Enumerated Constant type over the Simple Constant type. This optimization also implies that the order in which the *Classifier* queries each *Classification Type* class is significant.

As described in section 5.3.5, the *Fact Gatherer* and *Transformation Generator* actually perform the work delegated to them from their *Classification Worker* subcomponent's counterparts.

**Conflict Resolvers**

The *Conflict Resolvers* are helper libraries used by the *Classification Types*. Currently only two are necessary, though it is possible that as more classification types are implemented, more shared resolvers will be needed. Each *Conflict Resolver* is shared amongst all the various *Classification Types* as opposed to individual resolvers tailored for specific *Classification Type*s.

When trying to determine a declaration line for the macro, we run into conflicts such as those described in section 3.3. The *Declaration Line Resolver* examines an initial line and returns the closest line in which a macro may be safely declared, following the rules outlined in section 3.3.

Similarly, the *Name Conflict Resolver* takes the macro node and intended declaration line and returns a list of nodes that would conflict with the intended migrated declaration.

#### Classification Facts

This subcomponent is simply a data store of all the datum required to migrate the given macro type.

## 5.4 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. The *Source Transformer* is a process which uses the generated *Transform File* and actually migrates the code, as seen in figure 5.6.

Each transformation in the *Transform File* is processed individually. For the current file being processed, the *Source Transformer* skips to the line of the next transformation to perform. A list of the token nodes corresponding to this line is gathered from the facts and the transformation is performed on that list. We continue on to the next transformation so long as it affects the current line. Once we have exhausted all the transformations for the current line, the resulting list of tokens is converted back into plain text and written to the output source file.

## 5.5 Process View of Architecture

The described architecture consists of a number of processes and threads. To clarify the process view of the system, we present the following description.

As would be expected from their architectural diagrams and descriptions, the process views of the *Fact Extractor* and *Source Transformer* are a straightforward pipeline with each labelled

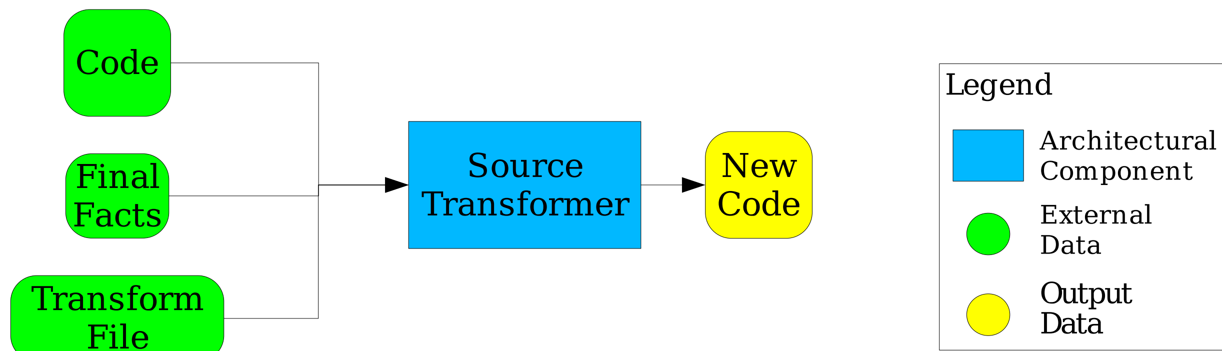


Figure 5.6: Source Transformer Component

architectural component being a separate process. The more interesting run-time architecture is in the *Migration Engine*.

Within the *Migration Engine* there are two processes, the *Driver* and the *Classification Worker*. Since the *Classification Worker* runs as a separate process and uses an XML-RPC interface, it is intended that each instance of it execute on a different system or cpu. Inside the *Driver*, the *Coordinator* is responsible for creating and managing the *Coordinator Threads* which actually communicate with the *Classification Workers*.

To help balance out any differences amongst the systems running the *Classification Workers*, a pool is kept which containing handles to each *Classification Worker*. This pool is called the *Handle Pool* in figure 5.7. Each *Coordinator Thread* chooses a handle from this pool every time it makes a RPC request, placing the handle back into the pool when the call is done.

An illustration of the process view for the *Migration Engine* can be seen in figure 5.7.

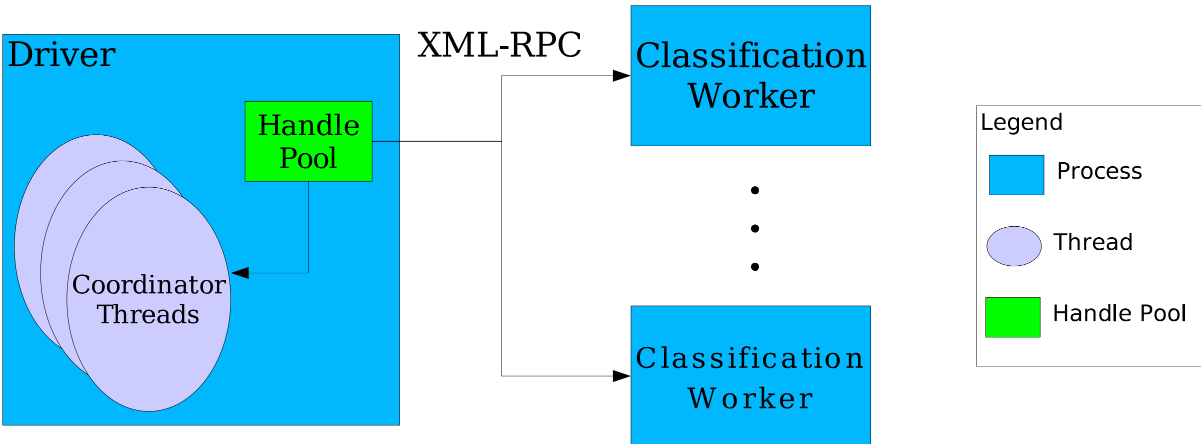


Figure 5.7: Migration Engine Process View

# Chapter 6

# Experimental Results

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. We intentionally chose an older version of Vim for its size. Both programs were built using a standard configuration for x86 Linux systems.

In the current system we have implemented six macro classifications: Simple Constants, Enumerated Constants, Constant Expressions, Parameterless Functions, Keyword Aliases, and Empty Declarations. Descriptions of these classifications can be found in section 4.2.

As mentioned in section 5.2, the fact extraction of the actual implementation deviated slightly from the method given in section 3.1 in that there was no macro dependency graph. As a result, only a single macro was migrated at a time, as opposed to potential groups of co-dependent macros.

Our experiment covered both sources and headers for each of Vim and Nethack. Due to their respective sizes we were able to use a single factbase for Vim, whereas Nethack required three. A rough limit of 100M was imposed on each factbase so as to ensure the facts and all the temporary

storage required for processing could fit within the author's personal PC's memory of 750M. The fact extraction was performed on a per C-file basis. Nethack's three factbases therefore contain no duplicate macro information for macros declared in C files, although they do contain duplicate information about macros declared in the headers.

The results of the experiment are summarized in table 6.1 and explained in table 6.2.

| **Metric** | **Vim** | **Nethack** |
|---|---|---|
| # of Files | 51 | 195 |
| KLoC | 26 | 108 |
| Declared macros | 493 | 3625 |
| Simple constants | 139 | 1822 |
| Enumerated constants | 193 | 648 |
| Constant expressions | 0 | 41 |
| Parameterless functions | 0 | 16 |
| Keyword alias | 1 | 2 |
| Empty declaration | 12 | 102 |
| Can not migrate | 0 | 15 |
| Ancestor declaration | 150 | 61 |
| Move to common scope | 0 | 0 |
| Decl. around first | 1 | 24 |
| Orig. interlacing decl | 0 | 0 |
| Name conflict | 4 | 4 |

Table 6.1: Results of experiment on Vim and Nethack

## 6.1 Discussion

From the summary we notice some interesting things. In both systems the relative number of constants is about 68%, which is close to what we would have expected, as given in the literature [5]. The number of declarations that needed to be migrated to an ancestor scoping block was surprising

at first.

Upon closer inspection we saw that in Vim, most of these macros were declared within a structure initialisation that defined editor commands. Inside this initialisation, 144 macros were declared after each initialized value as a sort of enumeration for each command entry. More-so, most of these entries were in fact migrated as Enumerated Constants.

Within Nethack we observed that the majority of these kinds of macro declarations were found in two structures. These structures described properties about monsters and objects within the game and the macro's declared within them were used as constants to describe various attributes.

The number of macros migrated to a common scope, being zero for both applications, indicates that they are fairly well behaved when it comes to taking advantage of the different scoping rules between C and CPP. This is also the case when considering the same number of macros whose declaration was interlaced with a C statement.

We would expect that naming conflicts indicate an error in the code. Upon further inspection however, it was found that a few constants were simply redefined with equivalent values. Furthermore, the number of macros that could not be migrated was extremely surprising. This should have meant that there were some serious problems with the code that could not be solved without significant refactoring. In actuality, a combination of bugs and restrictions in the algorithms used were the causes of these macros being marked as impossible to migrate. We further discuss the issue in section 6.2.

The remaining tables, from table 6.3 onward provide a detailed breakdown of each macro classification type and the migration metrics associated with it.

## 6.2 Confessions

The end result of running the implementation on Vim and Nethack should have been compilable sources that ran as the originals did. Unfortunately, while this was not the case, out of the 2859 macro migrations only 4 were incorrectly migrated.

### 6.2.1 Nethack

After some fixing by hand, we were able to get Nethack to compile. However, none of the problems encountered were unexpected or unknown. Due to time constraints a number of assumptions about the code given to the implementation were made. For example, it was assumed that a migrated macro would not ever be declared between a `goto` and the `goto`'s target, like the situation discussed in section 3.5. We found quite that a number of Constant Expression and Parameterless Function macros were dependent on other macros. Since there was no macro dependency graph, it is not surprising that the declarations of the migrated macros were not in the order they needed to be in to satisfy all the dependencies.

Only one such instance could not be resolved by simply rearranging the migrated declarations. The declaration for the function `Inhell()` in file `dungeon.h` needed to be moved into a separate file. Due to the dependencies between `dungeon.h` and `you.h` (which the function requires), there was no way to fix the code without some significant refactoring. The easiest change was to remove the function from `dungeon.h` and put it in it is own header file and then include the new header file where appropriate.

There were also an odd issue with implicitly casting between `char*` and `const char *`. Nethack frequently uses the conditional operator ”?” to determine things like which string to use

in a given situation. For example, in `eat.C`, the taste of food changes based on whether or not the player is hallucinating:

```
pline(Hallucination ?
        "Oh wow, like, superior, man!"
        :  "That food really hit the spot!")
```

After the migration process, a number of these kinds of conditionals required `(char*)` casts to be put in front of the strings, especially when a migrated macro was used for the string. However, it was not always necessary to do so and no obvious pattern for the problem emerged. Adding in the `(char*)` casts, where the compiler felt it necessary, was the simple solution to the problem.

It was very surprising to discover that, according to the system, some of the macros could not be migrated. It was later discovered that they were all erroneously marked as such. For some of these macros, the restrictive nature of the placement algorithm used in the implementation prevented their successful migration. Consider the case when the location of the original macro declaration is deemed to be a good spot for the migrated line. If that declaration is within a structure, for example, then we must relocate it. Currently, the system will only consider placing the declaration before the offending structure block. However, if the macro references the structure then the migrated line should be placed after the structure declaration. At this point in time, the system does not do this and consequently marked a number of macros in this situation as being impossible to migrate.

For the other erroneously marked macros, they were considered impossible to migrate due to bugs in the system. In some instances the Parameterless Function classification object was overly zealous and marked some macros as impossible to migrate, when in truth they should have been classified as a Parameterless Function With Variable Use (had that classification type object been implemented). As well, some of the macros were used as aliases to access member variables of

structures, which the system did not catch and ultimately decided were impossible to migrate.

### 6.2.2 Vim

Compared to Nethack, Vim was very well behaved and only two problems with it were encountered. The first problem was with a Function Alias macro that was classified as a Parameterless Function. Although the Function Alias macro classification was not implemented, the misclassification itself should not have caused a problem. In this particular instance, the return type of the migrated method was incorrect due to the system getting confused by the unusually complex return type for it. After some time spent trying to get the correct return type, the declaration was reverted back to the original preprocessor directive.

The final problem actually occurred during the source migration phase. In the file `term.h` there are a number of lengthy multi-line strings representing termcap entries. Due to a bug in the source migrator, we were not able to successfully migrate these macros.

| Metric | Explanation |
|---|---|
| # of Files | Number of C and header files in the application |
| KLoC | Number of uncommented source lines |
| Declared macros | Total number of declared macros |
| Simple constants | Macros classified as Simple Constants |
| Enumerated constants | Macros classified as Enumerated Constants |
| Constant expressions | Macros classified as Constant Expressions |
| Parameterless functions | Macros classified as Parameterless Functions |
| Keyword alias | Macros classified as Keyword Aliases |
| Empty declaration | Macros classified as Empty Declarations |
| Can not migrate | Identified number which can not be migrated, or can not be migrated without significant refactoring |
| Ancestor declaration | Migrated macros placed in a scoping block that was an ancestor of the original declaration. This occurs when the original declaration block will not allow the migrated declaration, such as in the middle of a struct (in C). |
| Move to common scope | Migrated macros that were declared in one C block, but used in a non-ancestor C block, requiring the migrated declaration to be located in a common ancestor block |
| Decl. around first | Migrated macros whose final declaration was made right before the first expansion |
| Orig. interlacing decl | Macros that were originally declared in the middle of a C statement |
| Name conflict | Migrated macros whose final declaration would have caused a name conflict with some other entity |

Table 6.2: Explanation of metrics

| Metric | Vim | Nethack |
|---|---|---|
| Simple constants | 139 | 1822 |
| In-place migration | 119 | 1740 |
| Ancestor declaration | 13 | 58 |
| Name conflict | 4 | 4 |

Table 6.3: Details for simple constant macros

| Metric | Vim | Nethack |
|---|---|---|
| Enumerated constants | 193 | 648 |
| In-place migration | 56 | 646 |
| Ancestor declaration | 137 | 2 |
| Name conflict | 0 | 0 |

Table 6.4: Details for enumerated constant macros

| Metric | Vim | Nethack |
|---|---|---|
| Constant expressions | 0 | 41 |
| In-place migration | 0 | 38 |
| Ancestor declaration | 0 | 1 |
| Name conflict | 0 | 0 |

Table 6.5: Details for constant expression macros

| Metric | Vim | Nethack |
|---|---|---|
| Parameterless function | 0 | 16 |
| In-place migration | 0 | 12 |
| Ancestor declaration | 0 | 0 |
| Name conflict | 0 | 0 |

Table 6.6: Details for parameterless function macros

| Metric | Vim | Nethack |
|---|---|---|
| Keyword alias | 1 | 2 |
| In-place migration | 1 | 2 |
| Ancestor declaration | 0 | 0 |
| Name conflict | 0 | 0 |

Table 6.7: Details for keyword alias macros

| **Metric** | **Vim** | **Nethack** |
|---|---|---|
| Empty declaration | 12 | 102 |
| In-place migration | 12 | 102 |
| Ancestor declaration | 0 | 0 |
| Name conflict | 0 | 0 |

Table 6.8: Details for empty declaration macros

# Chapter 7

# Conclusion

We have explored within this thesis 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.

We describe the operation and capabilities of our implemented system, 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.

## 7.1 Future Work

Despite the minor bugs encountered, the positive results of the case study encourage continuing development of the system and theory behind it.

Currently, our system only implements a few of the many macro types given in section 4.2. One of the obvious improvements would be to continue implementing the remaining macro classifications. In particular, attention needs to be given to the conditional preprocessor directives. Apart from the known difficulties in dealing with configuration management, the handling of conditional directives may have a significant impact on the process and architecture of the system. For example, consider the situation of conditional directives being migrated before other the macro types. In this case we would not need to worry as much about the placement of the other migrated macro declarations, at least when it came to ensuring that they were within the original conditional.

Very little time has been spent on trying to optimise the migration process. It could be beneficial to explore techniques in pre-selecting macros to migrate, with the intent of speeding up the overall processing time. The smaller the factbase used, the faster the processing will be. If we could select macros within small groups of files to migrate, instead of the entire system at once, we should be able to improve the overall migration time. Alternatively, some optimisations could be found in changes to the Grok interpreter. Currently, only a single environment is used by grok. What this means for us, is that it is difficult to isolate minimal sets of data to work with. It would be convenient if we could invoke multiple environments within Grok, with some means of transferring data amongst them. One environment could contain facts for the entire system, whereas another could consist of facts for a single file, for example. The flexibility in doing this would allow us more optimisation options with our data and data queries.

# Bibliography

[1] 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.

[2] 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.

[3] I. Baxter and M. Mehlich. Software Change Through Design Maintenance. In *International Conference on Software Maintenance*, pages 250–259, Bari, Italy, 1997.

[4] A. Cox and C. Clarke. Relocating XML Elements from Preprocessed to Unprocessed Code. In *10th International Workshop on Program Comprehension*, pages 229–238, Paris, 2002.

[5] Michael D. Ernst, Greg J. Badros, and David Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, December 2002.

[6] Jean-Marie Favre. The CPP Paradox. *http://citeseer.nj.nec.com/favre95cpp.html*, 1995.

[7] Alejandra Garrido and Ralph Johnson. Challenges of Refactoring C Programs. In *Proceed-*

*ings of the International Workshop on Principles of Software Evolution*, pages 6–14. ACM Press, New York, NY, 2002.

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

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

[10] Ric Holt, Tom Dean, and Andrew Malton. CPPX - C/C++ Fact Extractor CPPX. *http://swag.uwaterloo.ca/~cppx/*, January 2004.

[11] Y. Hu, E. Merlo, M. Dagenais, and B. Lagüe. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *International Conference on Software Maintenance*, pages 196–206, San Jose, California, 2000.

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

[13] P. Livadas and D. Small. Understanding Code Containing Preprocessor Constructs. In *3rd International Workshop on Program Comprehension*, pages 89–97, Washington D.C., 1994.

[14] 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.

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

[16] 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.

[17] H. Spencer and G. Collyer. `#ifdef` considered harmful, or Portability Experience with C News. In *USENIX Summer Conference*, pages 118–125, San Antonio, Texas, 1992.

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