Refactoring Browser with Preprocessor

Marian Vittek
Institute of Informatica at Comenius University
Mlynska dolina, Bratislava
Slovakia
e-mail:vittek@fmph.uniba.sk

Abstract

This paper presents Xrefactory a refactoring browser for C and Java languages. The paper discusses a way how refactoring browsers can thresh-passe non-trivial complications introduced by the C pre-preprocessor. Resolving those problems is the next step toward so long expected refactoring browser for C++. Our solution is based on practical implementation and exemplified on renaming refactorings.

1 Introduction

Refactoring [10, 6, 16] is a software development and maintenance process where the source code is changed "in such a way that it does not alter the external behavior of the code yet improve its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs" [16]. The word refactoring is used for the whole process of evolution, but also for its elementary steps. In this case *refactoring* (noun) stands for a well identified elementary and behavior preserving program transformation.

For example, renaming of a global variable iii to $fileIndex$ on all its occurrences is a refactoring. Replacing on all its occurrences means that only occurrences of this global variable will be renamed. There may be several local variables iii (or many structure records named iii) which will not be renamed because they are not *linked* to the global iii. In opposition to full text replacement this is a kind of *minimal* or *required* renaming, making only necessary modifications. Such refactoring improves the quality of the code because it makes it more readable.

At the first sight, refactoring does not appear as an important development concept. Usually, when a software works properly, users wish to add new features, not to maintain the old code. But, every developer knows that adding new feature to *badly written* software is difficult. In the same time, adding this feature to an equivalent software, but *well written* could be significantly easier. In the case you have a *badly written* legacy code and you need to add new features, the best way how to do it is turning your code into *well written* before doing anything else. Here is the moment when the refactoring appears.

Typical elementary refactorings are modifications like: renaming of variables and methods; moving of fields or methods from one class to another; moving of classes from one place to another; extraction of piece of code into new method; dead code removal; extraction of several fields into new class; turning variables into parameters; adding, removing or reordering method's parameters; etc.

Refactoring is not a new technique. Only the term and its formalization is recent. The term *refactoring* comes from the algebra and was first used in context of object oriented programming. As objects were seen as a factorization of data, restructuralization of objects was called refactoring. In this paper we use the term refactoring as a synonym for behavior preserving program transformation. We detach it from object oriented programming and we use it in context of classical imperative languages like C.

The point that the refactoring is behavior preserving is favorable to creation of automated refactoring tools. An *automated refactoring tool* is a development tool which is able to perform elementary refactorings without human interaction. It allows to maintain code by simply entering the name of the refactoring to be performed and its initial parameters. The particular refactoring is chosen from the collection of refactorings implemented by given tool. Refactoring tool then automatically modifies source code of refactored program. This requires in particular that the tool has global vision on the program. Refactoring tool must be able to resolve symbols in the source code, must be able to find all occurrences of given symbol, etc. Those functionalities are characteristic for source browsers and refactoring tools usually offers source understanding and browsing facilities. This is why they are called *refactoring browsers*. In the rest of the paper we will use this term for automated refactoring tools.

A refactoring browser should ensure that performed modifications are behavior preserving. Continuing our renaming example, a refactoring browser should check that the new name does not clash with an existing symbol. Of course, changes can not preserve the behavior in all aspects. The new program may be slower, for example. But the functionality of the new program should be the same. Conditions ensuring that a refactoring will be behavior preserving are called *preconditions* as they should be evaluated before application of refactorings.

Many research has been done in the refactoring area since its first investigations [10, 11, 13, 16]. In chronological order, refactoring started to be discussed mainly in the context of Smalltalk and C++ programming languages but without experimental implementations. Later, the Smalltalk refactoring browser [12] has been developed and was probably the world's first automated refactoring tool. The importance of refactoring was strengthen by *extreme programming* methodology [9, 2] as one of its basic rules. Independently, similar works on program transformations were held in the context of term rewriting [15]. Our work at Bratislava is connected to the term rewriting community in Nancy [3]. In 1997 we have started with development of a C and Java refactoring tool. After apparition of Martin Fowler's book [16] numerous implementations of refactoring browsers for Java emerged. However, the expected and logically very important tool for C++ is still missing. Practical benefits of C++ refactoring browser are obvious. Many very complex systems has been written in C++ and they require maintenance those days. Missing of an automatic refactoring tool is due to the complexity of the language itself, but also due to the pre-processor inherited from C. The preprocessor is not part of the language in traditional sense. It does not enter into its grammar and it makes standard techniques of refactoring very difficult to apply. We claim that pre-processor is the most important difference between C++ and other object oriented languages from the refactoring point of view. The importance of the pre-processor and a case-study of difficulties it introduces to the refactoring was presented in [8].

Refactorings were first defined for the maintenance of object oriented code. However, many of them can be applied also to simple languages like C. Many refactorings applicable on C language have been summarized in [7]. Despite the importance of the subject it seems that only few research is in progress in the area of C (or C++) refactoring browsers. There are several source browsers for C and C++, but in majority they do not have ambition to provide refactorings. In general, they are either based on regular expressions engine (which is insufficient for refactorings) or they act after preprocessing. When working with preprocessed sources many interesting informations are lost and the serious refactoring is practically impossible. Only few projects affront problems of the refactoring in presence of pre-processor. At Semantic Designs, there is a work in progress on a set of source understanding tools for variety of languages including C and C++ [1]. Alejandra Garrido and Ralph Johnson are working on a C refactoring browser at University of Illinois at Urbana-Champaign [7, 8]. From others, the work of Tokuda and Batory [14] at University of Texas on manipulation of C++ programs should be mentioned, even if they do not support preprocessor.

This paper presents our experiences with the refactoring browser named *Xrefactory* and shows one way how to refactor programs in presence of C pre-processor. The paper is also the reference paper for the Xrefactory project. The project is interesting because it is one of the first refactoring browsers for C. It was also recognized as the first refactoring tool implementing ExtractMethod refactoring for Java (and crossing Refactoring Rubicon) [5].

The main focus of this paper is pre-processor. Despite the first impression, problems related to pre-processor are not specific to C and C++. Also other languages are using constructions which can be considered as a kind of preprocessing. C# contains restricted macro definitions and conditional directives. Majority of programming languages allow comments to be intermixed into the source code. Comments can be inserted into the code without respecting any aspect of language grammar. They are not at the same level of processing (at the same processing phase) as the rest of the program. In this aspect they remain the preprocessor. In consequence several solutions discussed in this paper can be successfully applied to refactoring browsers for the majority of programming languages, not only those using C pre-processor.

The paper is organized as follows: the next section presents known aspects of refactoring browsers. Then, we present briefly the C preprocessor and section 4 presents how to extend general schema of refactoring browsers to incorporate pre-processing. Next sections explain in details how to manage different pre-processor constructions.

2 Automated Refactoring Tools

An automated refactoring tool is a software able to perform elementary refactorings. The selection of elementary refactorings is fixed and depends on the designers of the tool. Each elementary refactoring is described by an algorithmic description called *mechanics*. Mechanics is usually described in a generic way. For example: "traverse all references of given variable and replace old name by new one" is the mechanics for 'rename variable' refactoring. In some pathological cases, application of the mechanics onto a particular project may lead to unwanted errors. Following example demonstrates the pathological case when mechanical renaming of variable `i` to `val` changes program behavior.

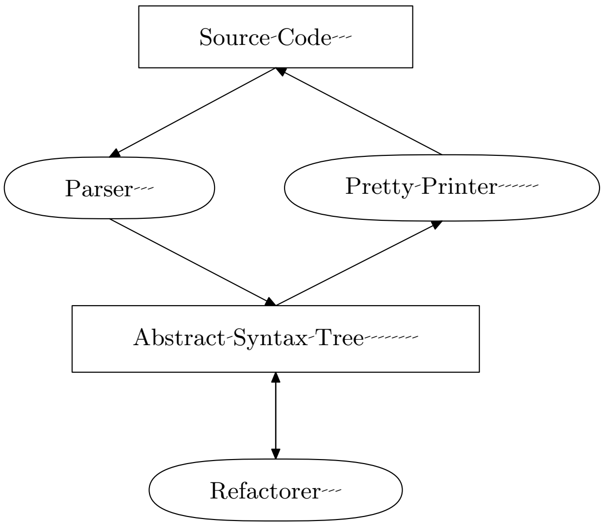

Figure 1. General schema of a refactoring browser.

```
static int i;
void seti(int val) {
    i = val;
}
```

As said previously, refactorings require global changes in the project and a refactoring browser needs to have overview about whole source code. In general it has two choices. Either to parse all files at each refactoring or to have an external data structure storing important informations and updated after each modification of source code. The second possibility is the only one reasonable for large projects and it is in general accepted in the design of refactoring browsers.

The usual overall architecture of a refactoring browser is shown in figure 1 and it covers: a *parser* for the given programming language transforming source code into an internal data structure (usually *abstract syntax tree*, shortly *AST*), the *refactorer* checking prechecks and applying refactorings on the tree and finally a *pretty printer* transforming the tree back to the source code.

A refactoring browser has to be based on full parsing. All attempts to base refactorings on simpler engine like regular expressions must fail for real languages, because programming languages are based on more complex structure. For example, variables have limited scopes and variables from inner scopes can override variables from outer scopes. Regular expressions are not able to capture those finesses. In consequence, the parser has to perform exactly the same work as a parser in a compiler. Otherwise the refactoring browser is too weak.

Refactorer is the main engine performing code modifications. It applies mechanics of available refactorings on the abstract syntax tree. Refactorer may provide *prechecks* to check for pathological cases. Tools not applying prechecks are often called *source modification tools* in opposition to *refactoring tools*. The latest should ensure 100% safetyness of effectuated modifications. Unfortunately, in practice, 100% safeteness can not be ensured without severe assumptions. In order to be practically usable, refactorers are not absolutely safe. However it is expected that possible sources of unsafeness are precisely reported. If user is aware of the danger and he confirms the operation, this one is applied even if potentially unsafe.

The last stage of a traditional refactoring browser is the *pretty printer*. It prints the refactored abstract syntax tree and replaces old code by the new one. Problems of pretty printing are obvious. Users do not agree that the pretty printer disorder their coding style. This concerns the code itself, but also those parts of code which do not enter into abstract syntax tree, like commentaries, and in our case preprocessor directives too. This problem is solved in traditional browsers by adding new informations into the abstract syntax tree. Those informations contain positions where the code is broken into lines, indentation offsets, positions and text of commentaries, etc.

The main focus of this paper is not on the standard techniques of refactoring browsers. Those are well studied for languages like Smalltalk and Java. Our focus is on preprocesor introducing irregularities.

3 Preprocessor

A usual C program is not written in C language. It contains *pre-processor directives* which are not part of C grammar. The source code written by user can not be parsed by a C compiler directly. Each source file is first passed to the pre-processor. Pre-processor looks for directives included in the text. Directives describe simple textual modifications which should be made at given place. After execution of those modifications a *pre-processed code* is generated. This code is then parsed and compiled by a C compiler into binary code.

The pre-processor enhances the expressive power of the language. The C language itself is very simple. Preprocessor gives him the power of modularity, symbolic names for constants, generic types and functions, possibility to maintain platform dependent code, etc. It is practically impossible to write and maintain a large application in C without using the pre-processor. Refactoring tools wishing to deal with C and C++ have no other alternative than accepting the pre-processor.

The exact definition of ANSI C pre-processor is beyond the scope of this paper. We suppose that reader is familiar with its basic ideas.

There are basically three pre-processing directives introducing problems to refactoring browsers. Inclusion of a file (#include directive), conditional compilation (#if - #else - #endif directive) and macro definitions (#define). Despite the very different nature of those directives, problems introduced to refactoring browsers are similar. In the rest of the paper we will present a way how to handle all those constructions in rather uniform manner.

4 Refactoring Tool with Pre-processor

In order to incorporate pre-processor into a refactoring browser one can take several approaches. The first approach consists in incorporating preprocessing directives into the language grammar. This is very direct way allowing application of traditional refactoring techniques on C. However, this way makes nearly all stages of the general refactoring scheme very complex. Since preprocessing directives may occur at any position in the program, they can break syntax structure of the language. Usually, the new language incorporating the original one together with pre-processor directives is not context free. It may be very difficult to create adequate grammar and a parser able to correctly parse such language. Also implementing refactoring upon generated parse tree may be very complex because usually the tree does not reflect the program structure, but structure of its text. It contains unexpanded macro calls, unevaluated includes and pre-processor conditional directives. The tree contains macro definitions where macro bodies contain unresolved symbols, because (following ANSI C standard) symbols inside macros are resolved according to the place where the macro is expanded, not according to place where the macro is defined. In summary, even if the approach incorporating pre-processor into C grammar seems to be very general and powerful it will be also very very hard to implement. As far as we know this is the approach undertaken in [1].

In opposition to the first approach, next two methods use standard, yet extended, pre-processor as presented for example in [4]. Such slightly extended preprocessor generates usual preprocessed code along with a function which maps positions from preprocessed code into their original positions in source code. Such preprocessor allows for any symbol from preprocessed code (which can be parsed by standard compiler) to identify its original position in the source code.

Usage of such pre-processor gives raised a second approach to development of refactoring browser. In this case, the source code is normally pre-processed and then parsed by standard C parser. For the purpose of refactoring additional informations generated by the preprocessor are stored into syntax tree. This approach has been taken in [7] and

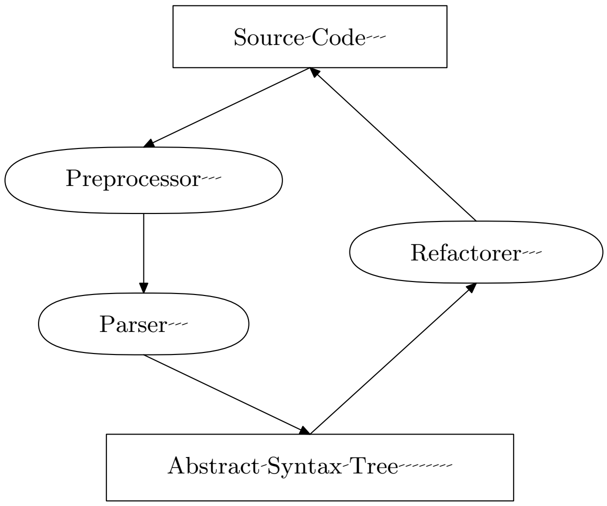

Figure 2. Xrefactory general schema.

it facilitates first two stages of refactoring browser. The parser is nearly the standard parser of the language and the refactorer operates on nearly standard syntax trees. However, the pretty printer will cause problems. It will need to reprint files included by #include directive into separate files and bodies of macros into original macro definitions as they were defined before expansion. In summary, pretty printed code has to respect the original organization of files and macro directives. Reconstitution of original program from its pre-processed (and refactored) syntax tree is the biggest difficulty (and challenge) of this approach.

Because of this difficulties, in Xrefactory, we have decided to use a third approach. Our approach is similar to the last one with the difference that refactorer does not apply changes onto abstract tree but directly on source code. This approach eliminates the necessity of pretty printer. The important point is that the preprocessed source code contains informations how it has been generated. Each piece of code contains information about its position in the original unpreprocessed code. The C parser understand those informations and incorporates them into abstract syntax tree. The overall architecture of our refactoring browser is shown in figure 2. Output of such refactoring browser is not a pretty printed modified code, but a list of patches (changes) needed to be applied to source code in order to obtain refactored code. Patches are in general simple editor directives like: "replace string $oldname$ by $newname$ in file $source.c$ at offset 532". This approach has several advantages. First, it completely eliminates problems with comments occurring in source code. In previous approaches comments cause problems as they need to be pretty-printed from AST, but they are not in the grammar of the language and refactoring browser needs to hack them into AST in some way. Using our approach, comments do not appear in AST. Second advantage is that changes performed on source code are local. Changes consist of a set of elementary modifications. Person using our tool can interactively confirm each elementary modification before it is applied. Especially persons starting to use refactoring tool, i.e. persons not very familiar with refactoring and not trusting in such automated tools appreciate the control over each performed modification.

However, also with our approach, the preprocessor still causes many problems. In the rest of the paper we will describe in details the problems we have encountered while developing Xrefactory. We will explain our difficulties step by step, starting with #include directive of pre-processor.

5 Include directive

The #include directive allows to inline content of a whole file into the compiled code. For example a fragment of code:

```
#include <stdlib.h>
```

says that in this place the whole content of the file `stdlib.h` is inserted and passed to the compiler.

Include directive looks very simply but, surprisingly, its correct handling is crucial for incorporating pre-processor into refactoring browser. As we will see in next sections, with a little bit of imagination, both macro expansions and #if directives can be handled using #include directive.

The #include directive divides source files into two (usually disjoint) parts. First, there are files which are send to the compiler directly as parameters. In the rest of this paper we will refer to them as *input files*. Second, there are files which are not entered to the compiler directly, but they are compiled as part of other files where they are *included*. We will refer to those files as *include files* or sometimes as *header files* because those files usually contain declarations (headers) of global functions. We will also use the term *compilation unit* to denote a single input unit entered to C compiler. Compilation unit has been yet preprocessed. It is usually an input file with all include files inlined and preprocessor directives expanded. A compilation unit is text written in pure C which can be directly parsed and compiled.

Standard suffix for header files is `.h` and for input files `.c`. In our following examples we will often remove suffixes when referring to files in order to make reasoning more fluent. So we will use description “header `a`” instead of “file `a.h`”. In general we will use letters `a,b,c,d` for header files and letters `u,v,w,x,y,z` for source files.

5.1 Linking symbols together

The main problem introduced by #include directive into Xrefactory is in putting together symbols defined in the same include file, but used in several compilation units. The problem comes from the fact that symbols are resolved and identified during parsing phase of source processing and in this phase sources are yet pre-processed and the information that symbols come from the same header file is hidden. But Xrefactory has to determine whether a symbol from one compilation unit refers to the same symbol as one from another compilation unit.

To illustrate the difficulty let us take following five source files `a.h, b.h, u.c, v.c` and `w.c`:

```
---------------------------- a.h
struct toto {
    int x;
};
---------------------------- b.h
struct toto {
    double x;
};
---------------------------- u.c
#include "a.h"
static struct toto y = {5};
---------------------------- v.c
#include "b.h"
static struct toto y = {5};
---------------------------- w.c
#include "a.h"
static struct toto y = {5};
----------------------------
```

producing three compilation units:

```
---------------------------- ~u.c
struct toto {
    int x;
};
static struct toto y = {5};
---------------------------- ~v.c
struct toto {
    double x;
};
static struct toto y = {5};
---------------------------- ~w.c
struct toto {
    int x;
};
static struct toto y = {5};
----------------------------
```

All compilation units can be compiled together into single executable program. In the example, two different structures ’toto’ are defined in two header files `a` and `b`. This situation does not cause any problem during compilation as no file is including both header files and structure names are considered local in compilation unit meaning they are not passed through compiler to linker. On the other side, a refactoring browser can not consider structure names as local for compilation unit, because for example in case of renaming they needs to be processed globally. In our example, the same structure `toto` is used in two compilation units `u` and `w` and one would very logically expect that a refactoring browser identifies those two structures. So, if user decides to rename this structure in the file `w` then the refactoring browser has also rename all its occurrences in files `a` and `u`. but not those in `b` and `v`. The renaming should lead to the code:

```
---------------------------- a.h
struct newName {
    int x;
};
---------------------------- b.h
struct toto {
    double x;
};
---------------------------- u.c
#include "a.h"
static struct newName y = {5};
---------------------------- v.c
#include "b.h"
static struct toto y = {5};
---------------------------- w.c
#include "a.h"
static struct newName y = {5};
----------------------------
```

When processing compilation units independently there is no evidence which structures should be renamed. In order to solve such problems we have enhanced the notion of *linking*. In the rest of the paper we will use term *linking* to express the way how different symbols are put together to be considered unique. We will also say that two symbols are *linked together* to express that they are considered equal by refactoring browser. There are several possible ways for a refactoring browser how to link symbols together.

The naive linking strategy would be to consider equal all symbols having the same name. In the rest of the paper we will refer to it as *name linking*. As we have seen on the example, the name linking does not correspond to the nature of C language. This is why we have introduced notion of *exact position linking*. Using this linking strategy two symbols are considered equal if and only if they have the same name and they are defined at the same position (line and column) of the same (header) file. This solution solves the problems from our example and is very natural. But, note that this way of identifying symbols is not 100% in the spirit of C language. ANSI C defines *type compatibility* as follows: two types are considered equivalent (compatible) in C language, iff they have the same name and they define the same type. For example two structures are considered to be same type iff they have the same name, the equal number of records and each record is compatible with its counterpart. There is no requirement on position where two structures are defined. For example let's take the following situation:

```
---------------------------- u.c
struct toto {
    int x;
};
static struct toto y = {5};

typedef int t;
struct toto {
    t x;
};
static struct toto z = {5};
----------------------------
```

This program compiles in ANSI C, no error about double definition of structure `toto` is issued. Both structures `toto` are of the same type, even if defined by two independent definitions. In ANSI C you can assign values of variable `y` to `z` and vice-versa. This leads us to define third way how two symbols can be linked together. We will refer to this way as *C compatible linking*. Using this kind of linking two symbols are considered equal if they have the same name and C compatible types. From the previous discussion one may think that exact position linking is stronger than C compatible linking in the sense that any two objects linked together by exact position strategy will be linked together by C compatible strategy. Also it may seem that C linking strategy is the best choice for a refactoring browser.

Following example demonstrates that both suppositions are false. Let's take following five files:

```
---------------------------- a.h
typedef int t;
---------------------------- b.h
typedef double t;
---------------------------- c.h
struct toto {
    t x;
};
---------------------------- u.c
#include "a.h"
#include "c.h"
static struct toto y = {5};
---------------------------- v.c
#include "b.h"
#include "c.h"
static struct toto y = {5};
----------------------------
```

In this example headers `a` and `b` define two different types, but both are named `t`. The header `c` is using this type for defining record `x` of structure `toto`. Source files `u` and `v` are then using this structure but after having included respectively headers `a` and `b`. In consequence, both source files are using the structure defined at the single place, but this definition defines two different types from the point of view of ANSI C. When renaming structure `toto` one would naturally expect that all occurrences in files `c, u` and `v` will be affected. However, in the sense of C compatible linking the structure used in file `u` is not the same as the one used in file `v`. In consequence renaming would not affect all those occurrences. In this example both exact position linking and simple name linking would succeed to all three occurrences of `toto` structure together and will allow correct symbol renaming, but C compatible linking doesn't.

From the previous discussion we can conclude that exact position linking is the most appropriate for refactoring browsers.

Making symbol resolution dependent on their position in the source code implies two problems. First, pre-processor needs to encode original position to each symbol and parser needs to use this information to create a unique *linking* name of the symbol for use in the Abstract Syntax Tree (AST) storing global informations about project. In this AST each symbol will be determined by its name and definition position. Second problem arrives when this AST needs to be updated after modification of only few source files. Let us considere the situation that the header file has been modified, say an empty line was added at the very beginning of the file. Then when taking any symbol defined in this file, the browser will not be able to find any reference of this symbol in the global AST. The AST will not contain symbol of the given name and position (because current position is increased by new line). Of course we can re-parse all files (including modified header file) and create AST from scratch. In such case everything will work fine, however in practical implementation such update of AST is undesired. Moreover not all kinds of program symbols needs to contain informations about definition place in order to be correctly linked together. For example global functions has to be unique in the project and hence are uniquely identified by their names.

Because of those practical reasons Xrefactory allows to restrict informations contained in AST. It offers two modes: the first one is strict position linking for situations requiring high precision of performed modifications (for example when renaming symbol with numerous definitions). The second mode is used during normal operation when information can be restricted in order to minimize necessity of updates of AST. In this second case information about position for structure names is restricted to file name. This allows to modify header file and the structure is still resolved correctly (unless the definition is moved into another file). Global functions and variables do not need position informations at all as their names are global (and unique) for the project. Unfortunately, in case of macros and local symbols the information can not be restricted.

5.2 Resolving ambiguities

In previous section we have solved problem arising when one single definition in a header file defines two distinct symbols. A similar but "opposite" problem occurs when two different symbols are used at the same place in a header file. Let's take the following situation:

```
---------------------------- d.h
static int listLen(struct list l) {
    int n=0;
    while (l!=NULL) {n++; l=l->next;}
    return(n);
}
---------------------------- x.c
struct list {
    int i;
    struct list *next
};
#include "d.h"
---------------------------- y.c
struct list {
    double x;
    struct list *next
};
#include "d.h"
----------------------------
```

In this example there is header file defining function `listLen` operating on parameter of type `list`. There are also two source files, each defining its own type `list` and then including the header file. In this example let's imagine that you decide to rename the type `list` in the file `x` to `intList`. The mechanics of the refactoring says that you need to replace old name by new name on all its occurrences. This includes occurrences in the header `d` but (according to position linking) not those in file `y`. However if you do this, sources do not compile anymore, as you will receive an undefined type while compiling file `y`.

This example illustrates that a refactoring browser has to check whether symbols entered into refactoring are unique, i.e. that there is no other symbol used at the same place in a header file.

6 Conditional compilation

Conditional compilation allows to include or exclude piece of code into/from the resulting code. It is similar to `if` statement of programming languages, but in fact it is much more complex from refactoring point of view. For example let us take a real fragment of code:

```
#include <stdlib.h>
#if defined(__WIN32)
#include <windows.h>
#include <direct.h>
#else
#include <dirent.h>
#endif
```

This code says that if the macro `__WIN32` is defined (i.e. when this program is compiled under MS-Windows platform) then standard headers `windows.h` and `direct.h` have to be loaded. Otherwise (a UNIX-like platform) standard header `dirent.h` will be loaded. Just to clarify, last two headers define nearly same functionalities, but using slightly different names.

This example demonstrates the standard usage of #if directive. It demonstrates that the #if directive is used when the same program is compiled in two very different environments. The #if directive can be (and often is) used in situations when the **if** statement from C language can not be used. It is clear that program including `windows.h` header will not compile under Unix platform and vice-versa. One can note that the file `stdlib.h` included in our example on both platforms refers in reality to two different files. When compiled with `__WIN32` macro predefined, this will be an MS-Windows file stored in directory with MS-Windows includes. The same directive however will refer to another include file when this source code is compiled under Unix system. In this case it will refer to file `/usr/include/stdlib.h`.

The first idea how to handle conditional directives may be some kind of backtracking over both branches. This would mean that source analyzer will parse the first branch of #if directive, then it goes back reconstructing state of parsing tables and then continuing by #else branch. A similar approach to this one is idea of elimination of #else directives by transforming each #if-#else directive into two successive #if directives, where the second condition is negation of the first one.

Another possibility is incorporation of #if directives into C grammar. Since usually #if directives do not break the structure of source code, this looks reasonable. However, real situation is more complex. Conditions may be complex and may rely one on another as well as on an extern environment of the compilation (as shown in the example of different include directories). Another problem of this approach is so called "#if 0" construction. This directive is often used to discard some part of source code from compilation. This part of source code may be syntactically incorrect, it may contain large commentaries. For example:

```
#if 0
This file with long comment /* perfectly */
compiles with any C compiler.
#endif
```

This is a correct C source code as the text violating C grammar is enclosed in #if 0 directive a hence is never passed to parser. Such cases can not be determined fully automatically, let's take the example:

```
#if ZERO
This file /* perfectly */ compiles unless
someone defines macro ZERO as non-zero.
#endif
```

In fact, the presence of the #if directive does not imply that there are two different ways how to parse source file. All possible ways how to process #if directives are given by external compiler environment fulfilled by user. Programmer writing those directives is aware about all possible processings.

From this opinion comes our approach of handling #if directives. We simply suppose that a compilation unit is given not only by input file, but also by its initial environment settings. I.e. by the initial setting of predefined macros and include directories. In consequence, each source file containing an #if directive may give two different compilation units. Hence, a source file containing n different #if directives may give 2^n compilation units. This may be too much for practical purposes. Fortunately, there are not very much different #if directives in a usual source code. Moreover not all combinations are reasonable. Usually only those combinations which are really used to compile sources are necessary to be explored. Those combinations can not be inferred automatically, so our solution is that user specifies them in a configuration file.

The solution used for linking together symbols from different compilation units then applies also on linking together symbols from the same input file but processed in several compilation units (with different initial environments). Our approach makes handling of #if directives uniform to handling of #include directive as it is using the same procedure for combining result of different passes through source code. One can imagine that our approach is transforming source files into equivalent ones which do not need to take care about multiple conditional branches.

For example let's take the program:

```
---------------------------- u.c
void printi(int i) {
#if defined(__WIN32)
    Jbox(i);
#else
    print("value is %d", i);
#endif
}
----------------------------
```

In order to process correctly such program, user has to specify two passes, one having `__WIN32` macro defined and second not. Based on this information, we then process the file two times as if there were two files:

```
---------------------------- pass1-u.c
#define __WIN32 1
#include "u.c"
---------------------------- pass2-u.c
#undef __WIN32
#include "u.c"
----------------------------
```

While processing files we do not care about #if directive, simply we let pre-processor to evaluate #if condition and discard one of branches:

7 Macro replacement

The pre-processor recognizes two forms of macro directives: *object-like* macros and *function-like* macros. Even if both forms are very similar, we will present them in separate sections.

7.1 Object like macros

Object like macros (also called identifier replacement) allow to define special identifiers (*macros*) and to postulate that those identifiers has to be replaced by the given text in the rest of the compilation unit. A typical object like macro definition is for example:

```
#define MAX_SHORT_INT 32767
```

Defining macro "MAX_SHORT_INT". This identifier has to be replaced by string "32767".

At the first sight it may seems that #define directive will cause serious problems to refactoring browsers. For example, just take the situation:

```
------------------------------- z.c
#define PRINTI printf("%d", i)
void fun1(int i) {
PRINTI;
}
void fun2(int i) {
PRINTI;
}
-------------------------------
```

This example defines two functions `fun1` and `fun2`. Both having one argument `i` and body containing expansion of `PRINTI` macro. This macro is printing the value of `i`. In this example if user decides to rename parameter `i` of function `fun1` to (say) `ri`, this will cause that `i` has to be renamed to `ri` also in the body of macro `PRINTI`. After this program does not compile anymore, since the function `fun2` also contains macro `PRINTI` and will now refer to undefined variable `ri`.

However this problem is very similar to the one we have solved in the section describing #include directive. Thinking deeper on the nature of both directives, we see that from the point of view of refactoring the #define directive is in some way similar to #include. We can transform all #define directives onto #includes by storing bodies of all macros into special headers, each macro body in one header file. Then any macro invocation is in fact just inclusion of the file containing its body. For example, here is the program equivalent from refactoring point of view to last example but without using #define directive:

```
--------------------------- printi-macro.h
printf("%d", i)
--------------------------- z.c
void fun1(int i) {
#include "printi-macro.h"
}
void fun2(int i) {
#include "printi-macro.h"
}
```

And since we know how to solve all problems introduced by #include directive, we can use those techniques to implement #define, simply by keeping in mind this program transformation. Of course, the real implementation will not generate all those temporary files with macro bodies. However, it will use the same routines for inlining macros and for including file.

7.2 Function like macros

Function like macros allow to define macros with parameters. This is an extension to previous form where, before each replacement of a macro by its body, formal parameters are replaced by actual. From refactoring point of view also this case can be transformed to #include directives as previously. The difference is that we need to generate a special header file for each invocation of the macro (not only for definition). Such generated header files will contain macro body with macro parameters replaced by their actual values.

The real problem with function like macros is introduced by ## operator. This operator permits to join two identifiers (at least one of which is macro parameter) giving raised new symbol. For example source code:

```
#define SETXY(v,a,b) {v##x = a; v##y = b;}

void main() {
    double point1x, point1y;
    double point2x, point2y;
    SETXY(point1, 3, 4);
    SETXY(point2, 5, 6);
}
```

Results after pre-processing into:

```
void main() {
    double point1x, point1y;
    double point2x, point2y;
    point1x=3; point1y=4;
    point2x=5; point2y=6;
}
```

Unfortunately, in this example if you decide to rename the variable "point1x" to (say) "x1", there is no way how to perform the renaming in a natural way. This problem is, in general, unsolvable and Xrefactory, for example, displays an error message. However, those problems can be solved for some interesting sub-cases. Definition of those sub-cases and partial solutions are subject of our further research.

8 Conclusion

In this paper we have been concerned by implementation of a refactoring browser for the C language. We have presented problems arising from the pre-processor and one possible solution how to solve them in the case of renaming refactorings. All concepts presented in the paper are based on real implementation of Xrefactory which is a plug-in to Emacs and XEmacs editors. The implementation consists of around 50 000 lines of code, majority of which (around 90%) is written in C and the rest in e-lisp language. Xrefactory has been exhaustively used to refactor itself during six years of its development.

Availability: Xrefactory is a proprietary software. Fully functional evaluational version is available for free download at http://www.xref-tech.com/xrefactory.

References

[1] Ira D. Baxter and Michael Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proceedings of WCRE 2001: Working Conference on Reverse Engineering*, Stuttgart, Germany, 2001. IEEE Computer Society Press.

[2] Kent Beck. *Extreme Programming explained*. Reading, MA, Addison Wesley Longman, Inc., 107108., 2000.

[3] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *Proc. of the First Int. Workshop on Rewriting Logic*, volume 4. Elsevier, 1996.

[4] Anthony Cox and Charles Clarke. Relocating xml elements from preprocessed to unprocessed code. In *Proceedings of IWPC 2002: International Workshop on Program Comprehension*, Paris, France, 2002.

[5] Martin Fowler. Crossing refactoring's rubicon. *www.martinfowler.com/articles/refactoringRubicon.html*, 2001.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[7] Alejandra Garrido. Software refactoring applied to c programming language. Master's thesis, University of Illinois, Urbana-Champaign, IL, USA, 2000.

[8] Alejandra Garrido and Ralph Johnson. Challenges of refactoring c programs. In *Proceedings of IWPSE 2002: International Workshop on Principles of Software Evolution*, Orlando, Florida, 2002.

[9] Richard Garzaniti, Jim Huangs, and Chet Hendrickson. Everything i need to know i learned from the chrysler payroll project. In *Conference Addendum to the Proceedings of OOPSLA'97*, 1997.

[10] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–25, July 1988.

[11] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1992.

[12] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4), 1997.

[13] Don Bradley Roberts. *Practical Analysis for Refactoring*. PhD thesis, University of Illinois, Urbana-Champaign, IL, USA, 1999.

[14] L. Tokuda and D. Batory. Evolving object-oriented architectures with refactorings. In *Conf. on Automated Software Engineering, Orlando, Florida*, 1999.

[15] Mark van den Brand, Paul Klint, and Chris Verhoef. Re-engineering needs generic programming language technology. *ACM SIGPLAN Notices*, 2(32):54–61, February 1997.

[16] Martin Fowler (with contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.