Minimizing Parsing when Extracting Information from Code in the Presence of Conditional Compilation*

Stéphane S. Somé
Timothy C. Lethbridge
School of Information Technology and Engineering (SITE)
150 Louis Pasteur, University of Ottawa, Canada
ssome@csi.uottawa.ca

Abstract

Exploring and understanding a software system require parsing its source code to extract necessary information. This task is complicated by the use of conditional compilation, a means offered by several programming languages for selective compilation of source code based on conditions. In this paper, we discuss the difficulties in parsing code with conditional compilation. We argue that the effective way to ensure the extraction of all meaningful information from a source file is to parse a set of versions of that file defined by conditional compilation, and describe an heuristic based approach for the minimization of such parsing.

1 Introduction

The goal of our overall research project is to improve the productivity of software engineers who are maintaining complex software systems [3, 4]. The purpose of the research described in this paper, is to design facilities that will allow code exploration and understanding, in the presence of considerable conditional compilation. To ensure our work is industrially relevant, we are working with a large telecommunications system developed by Mitel corporation. This is written in a proprietary language called Mitel Pascal and makes extensive use of conditional compilation. The software engineers who work with this system regularly add new features, fix problems and reengineer portions of it.

A conditional compilation directive involves a boolean expression followed by some source code. This source code is compiled only if its associated boolean expression evaluates to *true* during a pre-processing process. The evaluation of a condition depends on values assigned to conditional compilation variables.

*This work is supported by NSERC and Mitel Corporation and sponsored by the Consortium for Software Engineering Research (CSER).

Conditional compilation has several uses, including compiling versions of the software for different platforms and with different sets of features. Unfortunately, the use of conditional compilation may result in quite complex code [10, 2]; thus the tasks of program comprehension and reverse engineering can be very difficult. The Mitel system we are studying consists on 3460 files and uses 107 different conditions in conditional compilation directives.

When conditional compilation is used in a program, the actual file compiled is only one of many possibles versions of the source code – each determined by a particular setting of conditional compilation variables. The compiler does not care about the parts of the code that are not included in the particular compilation. However, program comprehension and reverse engineering are concerned with the understanding of *all* the information in a source code [3]. Effective tools should therefore be able to show all entities in a system along with information about which states of conditional compilation variables permit that entity to be considered by the compiler.

Program understanding and reverse engineering tools obtain their data by parsing source code. We must therefore ensure that such parsing can extract every detail in the code in presence of conditional compilation directives. Ideally we would write a parser that could accept un-preprocessed code and process it in one pass. This is, in general, not possible for three reasons:

Firstly, conditional compilation directives may be used in a such way that only a subset of them are *syntactically* consistent with each other. For example, we might have code such as:

```
a :=
#if PRODUCT1
        func1(
#endif
#if PRODUCT2
        func2(
#endif
        arg);
```

Clearly, valid code would not be produced if PRODUCT1 and PRODUCT2 were both set; however at least one of them must be set. It would probably be impossible to design a parser that could anticipate every programmer's trick such as the above.

Secondly, it is common to use directives for documentation purposes, which are, in fact not supposed to be parsed. An example follows:

```
#if PRODUCT1
#if SPECIALFEATURE
        Error: Special feature can only be incorporated into product 2.
#endif
#endif
```

Thirdly, conditional compilation is frequently used to 'comment-out' code that the programmer wants to preserve, for some reason (e.g. because it is only partially complete or

because the programmer is unsure about whether it is, in fact, safe to delete it). Such commented-out code will often have syntax errors. More details regarding the above difficulties will be discussed in the next section.

There are several well known program comprehension tools, e.g. Sniff+ [11], Source Navigator [8] commercial tools for code exploration, Rigi [6], Software Bookshelf [1] and GHINZU [5]. Due to difficulties such as those described above, these tools do not, in fact, consider conditional compilation: They present only the state of the system under one particular setting of conditional compilation variables. This hampers the ability of software engineers to use such tools, since they must know in advance which variables to set, and are not able to easily discover the effects of different settings. The main objective of the current research is to overcome this problem.

Another line of research that considers conditional compilation is source code configuration re-engineering. Examples of such work are Snelting [9] and Pearse et al [7]. Snelting's approach involves rediscovering code configuration structure by computing a concept lattice based on conditional compilation directives. The structure found may then be used to assess the code structure and help reorganizing it. Pearse and al show the conditional compilation structure of a source file by generating a graphical representation. They also compute some metrics (lines of input source code, lines of conditional compilation logic, number of conditional logic branches and number of preprocessor variables) in order to evaluate conditional compilation complexity.

As described above, our research objective is to parse source files to extract all the information. A straightforward approach would be to exhaustively consider all the possible versions induced by conditional compilation. This supposes that one knows all possible combinations of conditions that can be used for pre-processing, and then performing a separate parse for each. The approach also supposes that one pre-parses the code to extract all the possible conditions guarding conditional compilation directives in a file.

A problem with the above approach is the potential great number of parses. The essence of our technique is that we try to compute the minimum number of parses needed so as to have processed every valid statement in the source code, and hence to have extracted all the information in the file. In general, we find that the number of parses can be reduced to an acceptably low number, although we do not claim to be able to find the minimum number.

This next section discusses in more detail the difficulties of parsing code in the presence of conditional compilation. We then discuss the exhaustive parsing approach and approaches to reducing the number of parses based on heuristics.

2 Difficulty in parsing code with Conditional Compilation

Syntactically, a conditional compilation directive is similar to a space or a comment as it may appear everywhere in a source code except inside tokens. That makes impossible to design a usable grammar for unpreprocessed code.

The following is an example of a Mitel-Pascal program with conditional compilation directives. This is an artificial example used to motivate subsequent discussions.

```
PROGRAM srctest(input,output);                                              (1)
CONST                                                                       (2)
{%IF appl_d}                                                                (3)
  my_cmd_line =                                                             (4)
  {%IF load_config = lc_ss7} ss7_command_line; {%END}                       (5)
  {%IF load_config = lc_main} dumb_maintenance_terminal; {%END}             (6)
{%END}                                                                      (7)
index = 1;                                                                  (8)
{%IF icode_enabled}                                                         (9)
TYPE                                                                        (10)
      proc_record = RECORD                                                  (11)
              name: prog_name;                                              (12)
              procid: process_id;                                           (13)
      ENDREC;                                                               (14)
{%END}                                                                      (15)
VAR                                                                         (16)
{%IF icode_enabled} app_pid : proc_record; {%END}                           (17)
app_name : STRING[10];                                                      (18)
{%IF NOT icode_enabled} app_pid : INTEGER ; {%END}                          (19)
PROCEDURE treat_proc;                                                       (20)
BEGIN                                                                       (21)
{%IF appl_d AND (load_config = lc_ss7 OR load_config = lc_main)}            (22)
   treat_cmd(my_cmd_line);                                                  (23)
{%END}                                                                      (24)
{%IF NOT appl_d AND icode_enabled}                                          (25)
   This is an error. icode should not be enabled in this situation.         (26)
{%END}                                                                      (27)
   treat_name(app_name);                                                    (28)
ENDPROC                                                                     (29)
BEGIN                                                                       (30)
   treat_proc;                                                              (31)
ENDPROG.                                                                    (32)
```

Parsing this program without selecting appropriate parts by pre-processing is impossible because the code obtained by simply removing {%IF $\cdots$} and {%END} do not follow the language syntactical constructs. As an example the following constant declaration corresponds to the program lines 4 to 6. This code segment clearly do not correspond to a Mitel-Pascal legal constant declaration.

```
my_cmd_line =
        ss7_command_line;
        dumb_maintenance_terminal;
```

There is another difficulty with the fact that not all the variants of source code obtained by pre-processing may be wanted. In fact some combinations of conditional compilation conditions may be prohibited, and it is not exceptional that some directives be used for documentation and thus are not supposed to be parsed. An example of a such conditional compilation directive in the above program is between lines 25 and 27. This directive clearly shows that the intention of the programmer is to have a compilation error when both appl_d is undefined and icode_enabled is defined.

Since an invalid program cannot be completely parsed, we only want to consider variants of the source code that are syntactically correct. However, to do this, we need to know which combinations of conditions are valid and which are not.

3 Exhaustive parsing of code with Conditional Compilation

The straightforward way of extracting all the information in a source program that has conditional compilation is 1) to find all the combinations of conditional compilation variables and their possible values, and 2) to parse the source code repeatedly with each of these combinations. This will guarantee that each section of the source code has been parsed; although many sections will clearly be parsed many times.

For the remainder of the paper, when we use the word 'variable' we are referring to a conditional compilation variable unless otherwise specified. To be able to find all possible combinations of variables, requires us to first find the variables themselves. In some cases, they may be enumerated at a particular location in the source code but this cannot be relied on. In fact, variables are usually set using a compiler directive, e.g. the `-D` option in C. To find all variables that may effect the compilation of the code, therefore, we must 1) scan the code to detect all directives, and 2) extract the variables and their values from the directives.

Once we have found all the variables and values the exhaustive approach then involves constituting all the possible combinations of variables/values, and parsing the source file with each of them. The remainder of this sections describes this process in detail; discussion of optimizations is defered to subsequent sections.

3.1 Definitions

A condition guarding a conditional compilation directive is either an *atomic condition*, or a *complex condition* which consists on a set of *atomic conditions* related with operators AND or OR.

An *atomic condition* $\mathcal{C}$ is a triple <*Sign, Var, Abstr_Val*> with : *Sign* denoting the fact that the condition is *negated* or not (*Sign* = "+" or "-"), *Var* a variable and *Abstr_Val* an *abstract* value. Notice that the sign of a complex condition is propagated to its atomic conditions according to *De Morgan's* laws.

A variable may be any legal element in the *preprocessing* language under consideration which can be used as the left hand side of a boolean expression. In Mitel-Pascal it may be any expression. Note that valid variables in the preprocessing language may differ from valid variables in the underlying language.

An abstract value represents whatever a variable can be compared to in a condition. An abstract value may describe a single value or a set of possibly infinite values. We distinguish the following classes of abstract values.

- DEF,
- UNDEF and
- (*Comparator*,*Value*).

DEF and UNDEF are predefined values which denotes the fact that a variable is defined or not. As an example, variable `APPL_D` has the value DEF in `SRCTEST` line 3. A condition such as {`%IF Var`} is *true* only if the value of the variable `Var` is DEF.

An abstract value (*Comparator*,*Value*) includes a comparator; a relational operator in the language considered ("=", "in", "<>", "<", "<=", ">" and ">=" in Mitel-Pascal), and a value; an expression. (*Comparator*,*Value*) represents a set concrete values $\mathcal{V}$ such as the boolean expression "$\mathcal{V}$ *Comparator Value*" is *true* for all values $\mathcal{V}$. As an example, (>, 5) represent the set of all values greater than 5.

Given a set of variables $\{Var_1, \cdots, Var_n\}$, we define a *setting Vset*, as a set of pairs $\{(Var_1, Abstr\_Val_1), \cdots, (Var_n, Abstr\_Val_n)\}$ with $Abstr\_Val_i$ a possible abstract value of Var_i.

3.2 Algorithm for exhaustive parsing

Using the above definitions, we have defined an algorithm for exhaustive parsing of source code with conditional compilation directives. The algorithm, which will form the basis for optimizations to be described later, is as follow.
Given a source file $\mathcal{F}$:

1. Scan to extract all the atomic conditions of conditional compilation directives.

   This first scan gets all the {%IF *Conds*} in $\mathcal{F}$ with each *Conds* being atomic or complex. We extract all the atomic conditions <*Sign, Var, Abstr_Val*> in *Conds*.

   We then determine all the variables used in conditional compilation conditions of $\mathcal{F}$ and all their possible abstract values.

2. Constitute all the possible variable setting.

   Given all variables and their possible abstract values, we build the set of all the possible variable setting. We then obtain all the possibilities with which the file $\mathcal{F}$ can be pre-processed.

3. For each variable setting *Vset*, parse $\mathcal{F}$.

   We follow exactly the rules used for pre-processing. When a conditional compilation directive ({%IF *Conds*} in Mitel-Pascal) is found, we

   (a) evaluate *Conds* according to the abstract values in *Vset*

   (b) if *Conds* evaluate to *true*, we parse the statement in the conditional compilation directive (until the corresponding {%END} in Mitel-Pascal)

   (c) if *Conds* evaluate to *false*, we skip it.

   When an error occurs during the parsing, we mark *Vset* as `invalid`, stop the current parsing, and get the next setting.

All the information about $\mathcal{F}$ is stored in a database which is updated with the additional information obtained after each successful parsing.

3.3 Example

The following describes the application of this algorithm to the program `SRCTEST` listed earlier.

1. Extraction of all the conditions of conditional compilation directives.

   We obtain the following conditions:

   - {%IF appl_d}
   - {%IF load_config = lc_ss7}
   - {%IF load_config = lc_main}
   - {%IF icode_enabled}
   - {%IF NOT icode_enabled}
   - {%IF appl_d AND (load_config = lc_ss7 OR load_config = lc_main) }
   - {%IF NOT appl_d AND icode_enabled}

   Figure 1 shows all conditional compilation variables and their possibles values.

2. Constitution of the possible variable settings.

   We obtain 12 possible settings:

   #1 {(appl_d, UNDEF), (load_config, UNDEF), (icode_enabled, UNDEF)},

   #2 {(appl_d, UNDEF), (load_config, UNDEF), (icode_enabled, DEF)},

Variable	Abstract Value
appl_d	UNDEF DEF
load_config	UNDEF (=, lc_ss7) (=, lc_main)
icode_enabled	UNDEF DEF

Figure 1: SRCTEST conditional compilation variables.

#3 {(appl_d, UNDEF), (load_config, (=, lc_ss7)), (icode_enabled, UNDEF)},
#4 {(appl_d, UNDEF), (load_config, (=, lc_ss7)), (icode_enabled, DEF)},
#5 {(appl_d, UNDEF), (load_config, (=, lc_main)), (icode_enabled, UNDEF)},
#6 {(appl_d, UNDEF), (load_config, (=, lc_main)), (icode_enabled, DEF)},
#7 {(appl_d, DEF), (load_config, UNDEF), (icode_enabled, UNDEF)},
#8 {(appl_d, DEF), (load_config, UNDEF), (icode_enabled, DEF)},
#9 {(appl_d, DEF), (load_config, (=, lc_ss7)), (icode_enabled, UNDEF)},
#10 {(appl_d, DEF), (load_config, (=, lc_ss7)), (icode_enabled, DEF)},
#11 {(appl_d, DEF), (load_config, (=, lc_main)), (icode_enabled, UNDEF)},
#12 {(appl_d, DEF), (load_config, (=, lc_main)), (icode_enabled, DEF)}.

3. Parsing of `SRCTEST`.

   The program is parsed assuming each of the above settings. Settings #4 and #6 cause a parsing error because the conditional compilation directive between lines 25 and 27 is parsed.

The exhaustive parsing ensure that all the possible variants of a source code defined by conditional compilation are considered. There is however a problem with this approach as lot of parsing may be needed for a same file. As an example `SRCTEST` is parsed twelve times. More generally given a file $\mathcal{F}$ with $S_{var} = \{Var_1, \cdots, Var_n\}$ a set of all its conditional compilation variables, lets $\mathsf{nb\_vals}$ be a function such as $\mathsf{nb\_vals}(Var_i)$ is the number of possibles abstract values of the variable Var_i, the number of parsing of $\mathcal{F}$ is: $\prod_{i=1}^{n} \mathsf{nb\_vals}(Var_i)$. This problem may not be serious for small software systems since, we can arrange to have files parsed infrequently and in batch mode. However, the time and resources needed to parse large software systems (as in our case) may be quite significant.

The exhaustive parsing can be optimized by taking into account the facts that:

1. Our objective is to extract all the information from code files. Therefore situations where conditional compilation directives are skipped are not pertinent unless not doing a such skipping induce errors.

As an example, the conditional compilation directive between lines 25 and 27 is the only one in the SRCTEST program that must be skipped because it always cause an error.

2. Some directives may be independent enough to be parsed together even if their conditions cannot be logically true at the same moment.

   As an example, a syntactic parser can successfully parse the program SRCTEST with both conditional compilation directives at the line 17 and at the line 19. That is possible because the language syntax allows that. However the conditions guarding these directives can not be logically true at the same moment.

3. Erroneous situations can be skipped once detected.

   As an example, an error occurs in the SRCTEST program, when APPL_D is undefined and icode_enabled is defined. It is useless to parse with a setting that makes this condition becomes *true* more than once.

Using the above, we have developed an optimized multiple parsing algorithm which try to reduce the number of different parsings needed to extract all the information from source code. We describe this algorithm in the next section.

4 Optimized multi-parsing

The objectives of the optimizations are: (1) to have as much as possible conditional compilation directives as possible parsed simultaneously ,and (2) to avoid retrying erroneous conditions. The optimizations are based on relationships which help to split the set of all conditions guarding conditional compilation directives in a file, into subsets of syntactically consistant conditions that may be *true* simultaneously. In the following, we first describe these relationships and then the optimized algorithm.

4.1 Relationships between conditions

We use two relationships: a *dependency* relationship and an *incompatibility* relationship.

4.1.1 Dependency relationship

An atomic condition C1 *depends on* an atomic condition C2 if each directive D1 guarded by condition C1 either

1. is nested in a directive D2 guarded by condition C2, or

2. is also guarded by condition C2.

Figure 2 illustrates this relationship.

What is interesting about this relationship is that if an atomic condition C1 *depends on* an atomic condition C2, any directive with C1 is parsed only if both C1 and C2 are *true*.

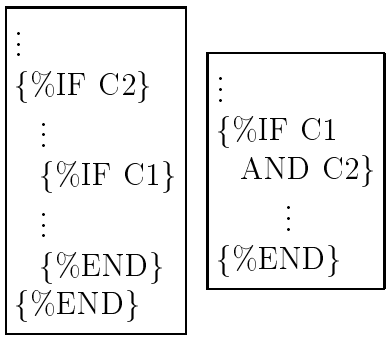

Figure 2: Situations where a condition C1 may depend on a condition C2. In these two examples, the directives with C1 can not be parsed when C2 is *false*.

In the program `SRCTEST`, condition `NOT APPL_D` (<-, APPL_D, DEF>) *depends on* condition `icode_enabled` (<+, icode_enabled, DEF>). Indeed, the single occurrence of `NOT APPL_D` is at the program line 25 in a condition where `icode_enabled` must also be verified.

4.1.2 Incompatibility relationship

In the following, we define a simple condition as:

- an atomic condition or
- an AND-combination of atomic conditions

Given that, a conditional compilation directive guarding condition is either:

- a simple condition or
- a OR-combination of simple conditions (in this last case, we consider that the corresponding directive is guarded by a set of simple conditions).

Two conditions, C1 and C2, are *incompatible* if a syntactic error occurs when both C1 and C2 are *true*.

This is a broad definition of *incompatibility* based on the assumption that there is at least one variant of the source file free of parsing errors. The assumption is reasonable since we are analyzing existing software systems. The definition of *incompatibility* corresponds to the following cases. Given a source file $\mathcal{F}$,

Case 1 An atomic condition C1 is *incompatible* with all the other conditions in $\mathcal{F}$ if there is a conditional compilation directive D1 not nested in any other directive, guarded by C1 such that the code within D1 causes a syntactical error. Figure 3 describes an example of a such case.

Case 2 A set of atomic conditions $\mathcal{C} = \{C_1, \cdots, C_n\}$ are incompatible with each others if there is a conditional compilation directive D1 not nested in any other directive, whose

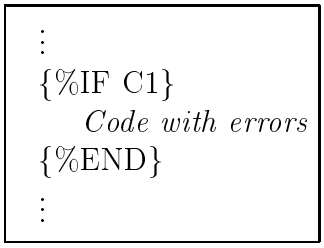

Figure 3: Situation of incompatibility corresponding to Case 1. C1 being an atomic condition, is incompatible with any other condition in the same source file.

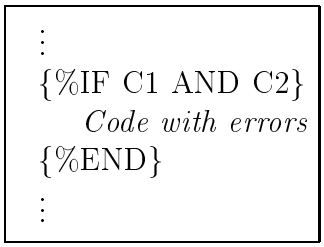

Figure 4: Situation of incompatibility corresponding to Case 2. Atomic conditions C1 and C2 are incompatible.

condition is the AND-combination of conditions in $\mathcal{C}$, and the code within D1 causes a syntactical error. Figure 4 describes an example of a such case.

There is an example of this kind of incompatibility in the program `SRCTEST`. Conditions <-, APPL_D, DEF> and <+, icode_enabled, DEF> are *incompatible* because the conditional compilation directive from line 25 to 27 guarded by the condition `NOT APPL_D AND icode_enabled` is syntactically incorrect.

Case 3 A simple condition C1 guarding a directive D1, is incompatible with a simple condition C2 guarding a directive D2 if D1 cause a syntactical error and D1 is nested in D2. Figure 5 describes an example of a such case.

Case 4 A simple condition C1 guarding a directive D1 is incompatible with a simple condition C2 guarding a directive D2 if there is a break in the syntax when the code in D1 and D2 are considered together. Figure 6 describes an example of such a case.

All these cases of incompatibility involve knowing that source code within directives cause errors. Moreover, in the fourth case knowing that there is an error does not help determine which conditions are incompatible because potentially any combination of directives (even distant in the code) may result in a break in the syntax. Strict incompatibility can therefore not be ascertained without syntactically analyzing the source file. However we have defined a weaker notion of incompatibility which allows us to use a heuristic approach to guess *potentially* incompatible conditions by a simple scanning of the source file. We also use a heuristic to deal with situations when parsing errors occur because of incompatible conditions.

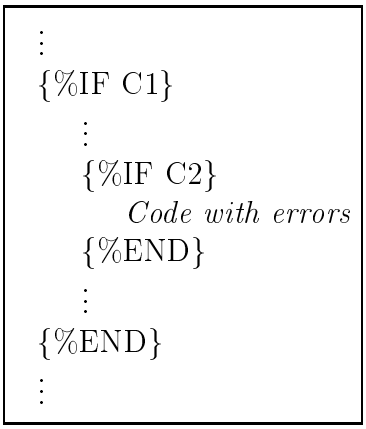

Figure 5: Situation of incompatibility corresponding to Case 3. Conditions C1 and C2 are incompatible.

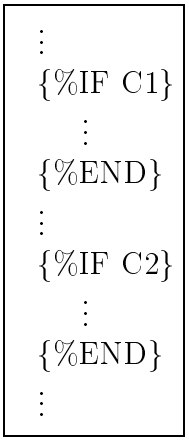

Figure 6: Situation of incompatibility corresponding to Case 4 if an error occurs when both C1 and C2 are true. Then, conditions C1 and C2 are incompatible.

4.2 Heuristics

We use two heuristics in the optimized multiple-parse algorithm. The first is used during an initial scan of the source file to determine conditions that may be incompatible during an initial scan of a source file; the second heuristic is used when a parsing error occurs to find the incompatible conditions which provoked it.

4.2.1 Potential incompatibility

Some incompatible directives (a subset of our fourth case of incompatibility) are used to define *case like* situations for source code inclusion. An example of a such use of incompatible directives in `SRCTEST` program, is the constant definition between lines 4 and 6.

This kind of incompatible conditional compilation directives directly follow[1] each other and are guarded by exclusive conditions. We use a heuristic to determine *potentially* incompatible conditions – a subset that covers most of the full set of conditions. The heuristic considers that conditions of conditional compilation directives which, (1) follow each other, and (2) use the same variables but different values or signs, are potentially incompatible.

This heuristic does not guarantee that all the incompatibilities between conditions are found nor that all potentially incompatible conditions are actually incompatible. The heuristic however allows us to deal with *case* situations such as that between lines 4 and 6 in the program `SRCTEST`. Moreover, potentially incompatible conditions may be found by scanning source files.

4.2.2 Dealing with errors

In section 4.1.2, we distinguished four cases of *incompatibility* between conditions that lead to parsing errors. The heuristic for finding potentially incompatible conditions defined above only encompasses conditions that correspond to the fourth case. It does not find incompatible conditions corresponding to other cases. Therefore, some errors may emerge when parsing with a given set of conditions. In a such a circumstance, it is not possible to determine with certainty which conditions are incompatible until we perform the parse. However, we use a heuristic such that when an error occurs, we can have some knowledge about the incompatibilities which led to that error. This heuristic is as follows:

When an error occurs during a parsing of a source file $\mathcal{F}$ with a particular subset of conditions $\mathcal{C}$,

1. Let Cerr be:
   - the guarding condition of D if the error occurred inside a conditional compilation directive D, or
   - the guarding condition of the latest directive D such that Cerr is in $\mathcal{C}$ if the error occurred outside of a conditional compilation directive.
2. Let CSet_last be:

[1] Two directives *directly follows* each other if there is no source code (except comments) between them.

- the set of all the guarding conditions of directives inside which D is nested if the errors occurred inside a conditional compilation directive D, or
- the empty set if the error occurred outside of a conditional compilation directive.

3. If Cerr is an atomic condition and CSet_last is empty, we consider that Cerr is incompatible with all the other conditions in $\mathcal{F}$ (this corresponds to the case 1 of incompatibility)

4. If Cerr is an AND-combination of conditions $Cerr_1 \cdots Cerr_n$ and CSet_last is empty, then each $Cerr_i$ is incompatible with the others (this corresponds to the case 2 of incompatibility).

5. If CSet_last is not empty then Cerr is incompatible with each of the conditions in CSet_last (this corresponds to the case 3 of incompatibility).

The heuristic does not ensure that we have found incompatibilities which cause all possible errors. However, our approach helps in situations such as that between lines 25 and 26, in the `SRCTEST` program.

4.3 An optimized multi-parsing algorithm

We have defined an optimized version of the exhaustive parsing algorithm. The principle of this optimization is to analyze as many conditional compilation directives as possible during a single parse. Starting with the set of all the conditions in a source file, the idea is to build subsets of *syntactically* compatible conditions using the dependency and the inconsistency relationships.

The dependency relationship is used to find conditions that it should be possible to make *true* simultaneously. If C1 *depends on* C2, there is at least one conditional compilation directive D that can not be parsed unless both C1 and C2 are true. Having C1 *true* and C2 *false* prevents us from parsing D. Thus, if D is ever to be parsed, C1 and C2 should hold together (and then be in a same subset), unless the designers actually intended that D never be parsed (we will ignore such cases).

Incompatible conditions must not hold together because that will lead to syntactical errors. If C1 and C2 are two incompatible conditions, we should ensure they are never in the same combination. But as previously discussed, it is not possible to ensure that a given set of conditions includes incompatibilities without actually performing parses that involve it.

We use a heuristic based on a weak definition of incompatibility to guess conditions that are potentially not compatible and prevent the system from considering them together. We also use a heuristic to deal with situations when parsing errors occur. Thus, the optimized multi-parsing algorithm is as follows:
Given a source file $\mathcal{F}$,

1. Scan $\mathcal{F}$ to extract CONDS, the set of all the conditions of conditional compilation directives, along with *dependence* and *potential incompatibility* relationships.

2. While there is a condition C not considered in CONDS

   (a) Let ACTIVE_CONDS = $\emptyset$

(b) add C into ACTIVE_CONDS

(c) add all conditions C *depends on* into ACTIVE_CONDS,

(d) add all conditions *depending* on C into ACTIVE_CONDS, if these conditions can be added without introducing incompatibilities into ACTIVE_CONDS,

(e) add into ACTIVE_CONDS all conditions of CONDS not yet considered and not *incompatible* with any condition of ACTIVE_CONDS

(f) parse the input file with all conditions in ACTIVE_CONDS set to be *true*

if there is an error, we apply our heuristics described above and so add more incompatibility relationships between some of the conditions in ACTIVE_CONDS.

if there is no error, we update the database and set all the condition in ACTIVE_CONDS as considered.

The above algorithm constructs a subset of active conditions from the set of all the conditional compilation conditions in $\mathcal{F}$. We build this set by considering as much as conditions possible, starting with an arbitrary condition C that is not considered yet. The set of active conditions thus include all the conditional compilation directives conditions except those incompatible with C. We do not check for incompatibilities when adding the conditions on which C depends (step 2-b) because of the weakness of our incompatibility relationship. Theoretically, a verification should be done to see if a condition is compatible with all its dependents.

4.4 Example

Consider the application of the algorithm to the program `SRCTEST`.

After scanning the program file, we obtain `CONDS` a set of all the conditions of conditional compilation directives in `SRCTEST`. Figure 7 shows these conditions with their *dependency* and *incompatibility* relationships as determined by the heuristic.

Condition	Depend on	Incompatible with
<+, appl_d, DEF>		<-, appl_d, DEF>
<+, load_config, (=,lc_main)>	<+, appl_d, DEF>	<+, load_config, (=,lc_ss7)>
<+, load_config, (=,lc_ss7)>	<+, appl_d, DEF>	<+, load_config, (=,lc_main)>
<+, icode_enabled, DEF>		
<-, icode_enabled, DEF>		
<-, appl_d, DEF>	<+, icode_enabled, DEF>	<+, appl_d, DEF>

Figure 7: Conditions in SRCTEST

In a second step, the algorithm builds subsets of consistent conditions (`ACTIVE_CONDS`) and parse `SRCTEST` with them. We build each of these subset around a condition not considered yet.

In the example, imagine <+, appl_d, DEF> is picked first. The algorithm adds this condition to `ACTIVE_CONDS` along with any condition that can be added without having incompatibilities in `ACTIVE_CONDS`. We obtain the set of conditions {<+,appl_d,DEF>, <+, load_config, (=, lc_ss7)>, <+, icode_enabled, DEF>, <-, icode_enabled, DEF>}. Notice that even if <+, icode_enabled, DEF> and <-, icode_enabled, DEF> are logically incompatibles, we consider them together because they are not syntactically incompatibles. Parsing with this first subset of conditions is successful.

The algorithm then looks for another condition not considered yet (e.g. <+,load_config, (=, lc_main)>) and constructs a new set `ACTIVE_CONDS`. We obtain {<+,appl_d,DEF>, <+, load_config, (=, lc_main)>, <+, icode_enabled, DEF>, <-, icode_enabled, DEF>} which also allows a successful parse of `SRCTEST`.

The algorithm then considers the condition <-,APPL_D,DEF> and generates the set `ACTIVE_CONDS` with {<-,appl_d,DEF>, <+,icode_enabled DEF>, <-,icode_enabled DEF>}. There is a parsing error with this setting because we parse line 26 of the program. We therefore apply our heuristic which sets conditions {<-,appl_d,DEF> and <+,icode_enabled DEF> to be incompatible.

Given this new information, the algorithm tries again to build a subset of the remaining conditions with which to parse the file. The system builds the subset around the condition <-,appl_d,DEF>. We obtain {<-,appl_d,DEF>, <-,icode_enabled DEF>} which allows a successful parsing.

We then stop parsing `SRCTEST` since all the conditions have been considered.

4.5 Limitation of the approach

The `SRCTEST` example shows that our objective to extract all the information in syntactically correct variants of the source code is still respected by the optimized multi-parsing algorithm because we parse the code inside all the conditional compilation directives except those that cause errors.

The example also shows that the algorithm allows us to reduce the number of parses that may be needed. In the example, we do four parses rather than twelve in order to consider all the conditional compilation directives.

However the approach is based on heuristics. Given the definition of incompatibility, we have defined a first heuristic to determine *potentially* incompatible conditions during an initial scanning process, and a second heuristic to find additional incompatibilities following a parsing error. These heuristics were motivated by the nature of incompatibility defined in section 4.1.2. Recall that we distinguish four cases of incompatibility. Our heuristics help finding incompatibilities that correspond to the three first cases and some incompatibilities in the fourth case.

There is however a subset of the fourth kind of incompatibility that can not be appropriately found using our algorithm because their directives are distant in the source code. For this problem to arise, it is sufficient to have in the language rules such as following (this kind of rule can be found in Mitel-Pascal).

```
R    ->  A B
      |  B C
```

Where `A`, `B` and `C` can be terminals or nonterminals. As an example consider a language with rule `R` and with `A`, `B` and `C` defined as follow:

```
A    ->  ``a''
B    ->  ``b''
C    ->  ``c''
```

Figure 8 shows an example of code with conditional compilation corresponding to the rule `R` and the three program variants that may correspond to this code.

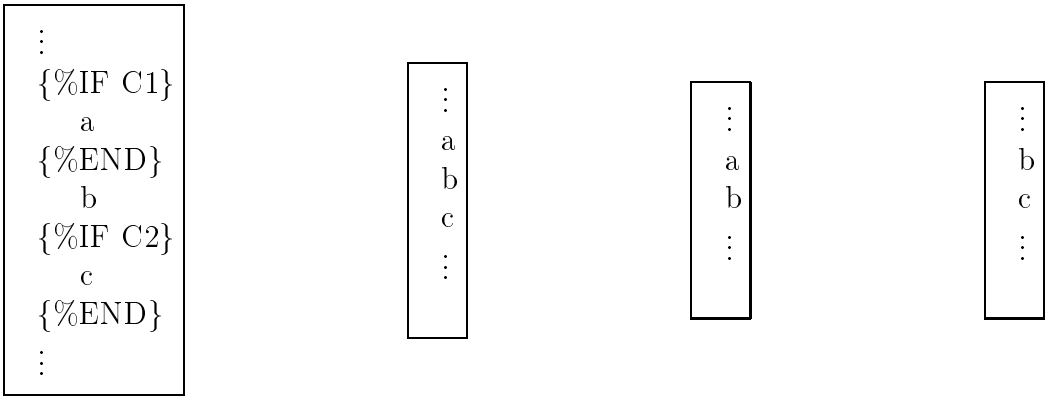

Figure 8: Program and its variants.

Conditions *C1* and *C2* are incompatible since a syntactically erroneous program corresponds to the case where both are *true*. However our heuristics incorrectly determine the incompatibility because:

- *C1* and *C2* do not directly follow each other and
- after the syntactical error while parsing with both *C1* and *C2*, we end up having:
  - if *C2* is an atomic condition not nested in another directive, *C2* incompatible with all the other conditions in the source file (not only *C1*), or
  - if *C2* is AND-combination of atomic conditions not nested in another directive, all the atomic conditions that make up *C2* incompatible with each other, or
  - if the directive guarded by *C2* is nested in other directives, *C2* incompatible with the guarding conditions of all these directives.

Notice that we still have a successful parse of the source file. However, our objective to consider all the information is not longer respected because of a directive being wrongly skipped.

The difficulty with this kind of incompatibility is that the directives involved may be arbitrary distributed in the source file. In fact, the rule `R` can be a non terminal rule, and therefore there may be several conditional compilation directives between the two incompatible ones.

5 Conclusion

Conditional compilation is a powerful tool that helps in developing multiple versions of a software system. All these version should be taken into account by browsing or code comprehension tools working on the source code of this software system. Therefore, the parsing process used to extract the relevant information from code source should consider conditional compilation directives. We have developed an exhaustive approach for to parsing such a program but because of its potential cost, we have proposed an optimized version of this approach. The optimizations are based on relationships between conditions found using heuristics.

Tests with a software system consisting of Mitel-Pascal files, shows that there is a real gain with the optimized approach. Using the exhaustive approach, the average number of different parses of each file having conditional compilation is 3.75. This average number drops to 1.29 with the optimized version.

The optimized approach however may fail for certain source files with a particular use of conditional compilation directives. That is, we can not ensure for these files that our objective to extract all the information is met. In fact, we can only guarantee that this objective is satisfied if there is no error when parsing the input file. This is the only situation where we are sure that the file considered does not fall in the class of files that we are not able to parse accurately. Finding a solution for this problem is still open. We suggest using the optimized approach as long as there are not errors and parsing problematic files using the exhaustive approach. The burden of this solution might be low because of the number of such problematic files. In our example, only three files cause parsing errors and the average number of parses with this approach goes from 1.29 to 1.47.

Thanks

The authors would like to thank all the members of the KBRE research group Nicolas Anquetil and Jelber Sayyad-Shirabad for fruitful discussion we have had with them.

References

[1] R. Holt. Software Bookshelf: Overview And Construction. http://www.turing.toronto.edu/~holt/ papers/bsbuild.html.

[2] K. Jameson. *Multi-Platform Code Management*. O'Reilly & Associates, Sebastopol,CA, 1994.

[3] T. C. Lethbridge and N. Anquetil. Architecture of a Source Code Exploration Tool: A Software Engineering Case Study. Technical Report TR-97-07, University of Ottawa, Computer Science, December 1997.

[4] T. C. Lethbridge and J. Singer. Strategies for Studying Maintenance. In *Workshop on Empirical Studies of Maintenance*, pages 79–84, Monterey, California, November 1996.

[5] P. E. Livadas and D. T. Small. Understanding code containing preprocessor constructs. In *3rd Workshop on Program Comprehension*, pages 89–97, November 1994.

[6] H.A. Müller, M.A. Orgun, S.R. Tilley, and J.S. UHL. A Reverse-engineering Approach to Subsystem Structure Identification. *Software Maintenance: Research and Practice*, 5:181–204, 1993.

[7] T. T. Pearse and P. W. Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In IEEE Computer Society, editor, *Proc. International Conference on Software Maintenance*, pages 270–277, 1997.

[8] Power Software Corporation home page. http://www.power-soft.co.uk/.

[9] G. Snelting. Reengineering of configurations based on mathematical concept analysis. *ACM Transactions on Software Engineering and Methodology*, 5(2):146–189, April 1996.

[10] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience With C News. In *Proc. Summer'92 USENIX Conference*, pages 185–197, June 1992.

[11] Take5 Corporation home page. http://www.takefive.com/index.htm.