Parsing Minimization 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 requires extracting meaningful information from it. This in turn involves syntactical analysis of source code, an activity that can be complicated by the use of conditional compilation.

In this paper, we discuss difficulties when 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. We then describe a heuristic-based approach to minimize the amount of parsing.

1. Introduction

The goal of our overall research project is to improve the productivity of software engineers who are maintaining complex software systems [4, 5]. 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 re-engineer 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 pre-processing. 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 [9]. Unfortunately, the use of conditional compilation may result in quite complex code [11, 2]; thus the tasks of program comprehension and reverse engineering can be very difficult.

When conditional compilation is used in a program, the actual file compiled is only one of many possible 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. In contrast, program comprehension and reverse engineering are concerned with understanding *all* the information in the code [4]. Effective tools should therefore be able to show all entities in a system along with information about which states of conditional compilation variables permit each 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 unpreprocessed code and process it in one pass. This is, in general, not possible for three reasons. Firstly, conditional compilation directives may be used in such a way that only a subset of them are syntactically consistent with each other. Secondly, it is common to use directives for documentation purposes, which are, in fact, not supposed to be parsed. 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+ [12], Source Navigator [8], Rigi [7], Software Bookshelf [1] and GHINZU [6]. 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.

The next section discusses in more detail the difficulties of parsing code in the presence of conditional compilation. We then discuss an approach that uses several heuristics to minimize the number of parses needed to process every valid statement in the source code, and hence to extract all the information in the file.

2. Difficulty in parsing code with Conditional Compilation

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

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

Syntactically, a conditional compilation directive is similar to a space or a comment as it may appear anywhere in the source code except inside a token. This makes it impossible to design a usable grammar for unpreprocessed code. Parsing the above program without selecting appropriate parts by pre-processing is impossible because the code obtained by simply removing {%IF $\cdots$} and {%END} does not follow the language's syntactical constructs. As an example, the following constant declaration corresponds to lines 4 to 9 of the program. This code segment does not correspond to a legal Mitel-Pascal constant declaration.

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

In addition to the above, the programmer may have *intended* that certain variants of the source code never be compiled. An example of this in the SRCTEST program is between lines 33 and 35. 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.

A straightforward approach would be to exhaustively consider all the possible versions induced by conditional compilation. This approach which is described in [10], supposes 1) finding all the combinations of conditional compilation variables and their possible values, and 2) parsing 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. There is however a problem with this approach: an excessive amount of parsing may be needed for each file. As an example SRCTEST is parsed twelve times. More generally given a file $\mathcal{F}$ with $S_{var} = \{Var_1, \cdots, Var_n\}$ (the set of all its conditional compilation variables), let nb_vals be a function such that nb_vals(Var_i) is the number of possible abstract values[1] of the variable Var_i; then the number of parses of $\mathcal{F}$ is: $\prod_{i=1}^{n}$ nb_vals(Var_i).

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

1. 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 parser that only considers syntax can successfully parse the program SRCTEST including both conditional compilation directives at line 23 and at line 27, despite the fact that the conditions cannot be logically true at the same moment.

2. 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 this condition *true* more than once. Without this optimization, multiple attempts at parsing with this condition may be made because the condition may be part of several more complex conditions.

Using the above, we have developed an optimized multiple parsing algorithm which tries to reduce the number of parses needed to extract all the information from a source file. We describe this algorithm in the next section.

3. Optimized multi-parsing

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 consistent conditions that may be true simultaneously. In the following, we first provide some definitions, then we describe these relationships and the optimized algorithm.

[1] We define abstract value in the next section.

3.1. Definitions

A condition guarding a conditional compilation directive is either an *atomic condition*, or a *complex condition* consisting of a set of *atomic conditions* related with operators AND (conjunction) or OR (disjunction).

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 (defined below). Notice that the sign of a complex condition is propagated to its atomic conditions according to De Morgan's laws [3]. 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 denote 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 which is a relational operator in the language considered ("=", "in", "<>", "<", "<=", ">" and ">=" in Mitel-Pascal), and a value which is an expression. (*Comparator*,*Value*) represents a set of concrete values $\mathcal{V}$ such that the boolean expression "$\mathcal{V}$ *Comparator Value*" is *true* for all values $\mathcal{V}$. As an example, (>, 5) represents the set of all values greater than 5.

We define a *simple condition* as:

- an atomic condition or
- a conjunction of atomic conditions

The guarding condition of a conditional compilation directive is then either:

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

Note that any complex condition can be rewritten as a disjunction of simple conditions using distributive laws [3].

3.2. Relationships between conditions

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

3.2.1 Dependency relationship

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

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

Figure 1 illustrates this relationship.

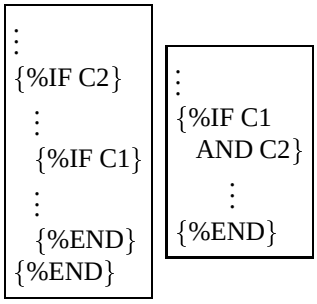

Figure 1. Situations where a condition C1 may depend on a condition C2. In these two examples, the directives with C1 cannot be parsed when C2 is *false*.

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*.

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` at the program line 33 is in a condition where `icode_enabled` must also be verified.

3.2.2 Incompatibility relationship

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 (and working) software systems. This 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 2 describes an example of such a case.

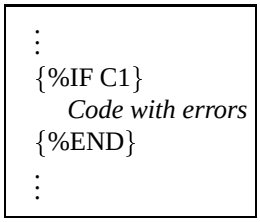

Figure 2. Situation of incompatibility corresponding to Case 1. C1 being an atomic condition, is incompatible with any other condition in the same source file.

Case 2 A set of atomic conditions $\mathcal{C} = \{C_1, \cdots, C_n\}$ are incompatible with each other if there is a conditional compilation directive D1 not nested in any other directive, whose condition is the conjunction of atomic conditions in $\mathcal{C}$, and the code within D1 causes a syntactical error. Figure 3 describes an example of such a case.

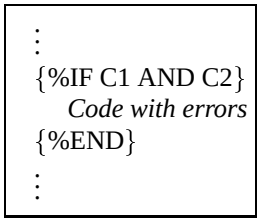

Figure 3. Situation of incompatibility corresponding to Case 2. Atomic conditions C1 and C2 are incompatible with each other.

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 33 to 35 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 causes a syntactical error and D1 is

nested in D2. Figure 4 describes an example of such a case.

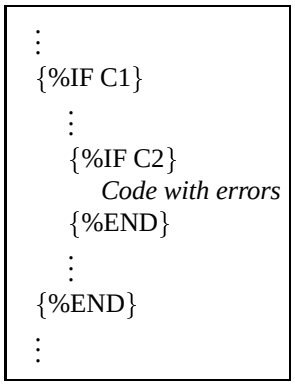

Figure 4. Situation of incompatibility corresponding to Case 3. Conditions C1 and C2 are incompatible.

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 parsed together while there isn't when only one of the directives is considered.

Figure 5 describes an example of such a case.

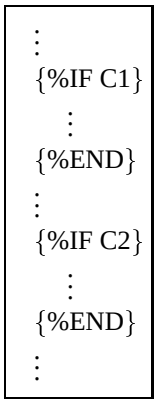

Figure 5. Situation of incompatibility corresponding to Case 4 if an error occurs when both C1 and C2 are true, then conditions C1 and C2 are incompatible.

All these cases of incompatibility involve knowing that source code within directives causes 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 [10]. 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 some simple scanning of the source file. We also use a heuristic to deal with situations when parsing errors occur because of undetected incompatible conditions.

3.3. 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; the second heuristic is used when a parsing error occurs to find the incompatible conditions which provoked it.

3.3.1 Potential incompatibility

A good proportion of incompatible directives (a subset of our fourth case of incompatibility) are used to define *case like* situations for source code inclusion. An example of such a use of incompatible directives in the SRCTEST program, is the constant definition between lines 4 and 9.

Several such incompatible conditional compilation directives directly follow[2] each other and are guarded by exclusive conditions. We use a heuristic to determine such *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 9 in the program SRCTEST. Moreover, potentially incompatible conditions may be found by quickly scanning source files rather than full parsing.

3.3.2 Dealing with errors

In section 3.2.2, 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

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

emerge when parsing with a given set of conditions. We use a heuristic such that when an error occurs, we can get some knowledge about the incompatibilities which led to that error. This heuristic, based on the definition of incompatibility, 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:
   - 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 a conjunction of atomic 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).

This heuristic helps in situations such as that between lines 33 and 34, in the SRCTEST program.

3.4. 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 cannot 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 the 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.

We use the two heuristics defined below to guess conditions that are potentially not compatible and prevent the system from considering them together, and to deal with situations when parsing errors occur. We also suppose that the parsing information extracted is stored in some "database". 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 that 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 in section 3.3.2 and so add more incompatibility relationships between some of the conditions in ACTIVE_CONDS.

   If there is no error, we update the database with the information obtained by parsing, and set all the conditions 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 simultaneously considering as many conditions as possible, starting with an arbitrary condition C that is not considered yet. The set of active conditions thus includes all conditions except those incompatible with C.

3.5. 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 6 shows these conditions with their *dependency* and *incompatibility* relationships as determined by the heuristic.

Condition	Depends 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 6. Conditions in SRCTEST

In a second step, the algorithm builds subsets of consistent conditions (ACTIVE_CONDS) and parses SRCTEST with them. We build each of these subsets 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 though <+, icode_enabled, DEF> and <-, icode_enabled, DEF> are logically incompatible, we consider them together because they are not syntactically incompatible. 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 ACTIVE_CONDS set. 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 34 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.

3.6. Application of the approach

We are doing this research with a large telecommunications system developed by Mitel corporation. This system has about 1.5 million lines of Mitel-Pascal code distributed in 3460 files. It makes extensive use of conditional compilation; there are 107 different conditions used to guard conditional compilation directives.

Tests with this software system show that there is a real gain with the optimized approach. Using the exhaustive approach, the average numbers of parses of each file, needed to consider all the conditional compilation directives and then extract all the information used for code exploration and understanding is 3.75. This average drops to 1.29 with the optimized version.

4. Conclusion

Conditional compilation is a powerful tool that helps in developing multiple versions of a software system. All these versions should be taken into account by browsing or code comprehension tools working on the source code of the software system. Therefore, the parsing process used to extract the relevant information from code source should take conditional compilation directives into account.

A direct way of doing that would be to exhaustively consider all the possible combinations of conditions that can be used for pre-processing, and then performing a separate parse for each. Because of the potential great number of parses needed with this approach, we propose an optimization based on heuristics. The application of our approach to a large software system shows a real gain in performance.

Our approach has been developed for Mitel-Pascal. However, the relationships and heuristics on which it is based can be adapted to other languages such as C or C++,

since the semantics of conditional compilation used in these languages are the same as in Mitel-Pascal.

Acknowledgements

The authors would like to thank all the members of the KBRE research group, expecially Nicolas Anquetil and Jelber Sayyad-Shirabad, for fruitful discussions 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] R. Kowalski. *Logic for problem solving*. New York : Elsevier North Holland, 1979.

[4] 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, Dec. 1997.

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

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

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

[8] Power Software Corporation home page. http://www.powersoft.co.uk/.

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

[10] S. S. Somé and T. C. Lethbridge. Minimizing Parsing when Extracting information from Code in the Presence of Conditional Compilation. Technical Report TR-98-01, University of Ottawa Computer Science, Jan. 1998.

[11] 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.

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