Handling Preprocessor-Conditioned Declarations

Lerina Aversano[*], Massimiliano Di Penta[*], Ira. D. Baxter[**]
{aversano, dipenta}@unisannio.it idbaxter@semdesigns.com

{*} RCOST- Research Centre on Software Technology
University of Sannio – Department of Engineering - Piazza Roma, I-82100 Benevento, Italy
{**} Semantic Designs Inc. - 12636, Research Blvd., C214, Austin – TX 78759-2239 - USA

Abstract

Many software systems are developed with configurable functionality, and for multiple hardware platforms and operating systems. This can lead to thousands of possible configurations, requiring each configuration-dependent programming entity or variable to have different types. Such configuration-dependent variables are often declared inside preprocessor conditionals (e.g., C language).

Preprocessor-conditioned declarations may be a source of problems. Commonly used configurations are type-checked by repeated compilation. Rarely used configurations are unlikely to be recently type checked, and in such configurations a variable may have a type not compatible to its use or it may contains uses of variables never defined.

This paper proposes an approach to identify all possible types each variable declared in a software system can assume, and under which conditions. Inconsistent variable usages can then be detected for all possible configurations. Impacts of preprocessor-conditioned declaration in 17 different open source software systems are also reported.

Keywords: Preprocessor code analysis, type-checking, symbol table, multi-platform software.

1. Introduction

Software systems are often developed with configurable capabilities, and to run under several different hardware and software platforms. Preprocessor conditionals are often used to configure the software accordingly. Setting the configuration variables and compiling gives a single configured instance of the program, and under ideal circumstances, each configurer can do just that and obtain a working program for his configuration.

However, full configurability is difficult to achieve in practice, and it can be impractical for the programmer to verify that every configuration works correctly. Knowing that every possible configuration is type-correct is a necessary first step to ensure that every configuration actually works. Testing a particular configuration for type-correctness is easily done if one has a compiler, by simply configuring and compiling. This approach is not effective once the number of configurations becomes significant. What we would like ideally is a tool to verify that a program is type-correct in all possible configurations.

This paper reports on initial steps towards creating such a tool, by defining an approach that can capture what types are declared under what configurations, and exploring how these conditional types can be used to detect configuration-instance type faults.

The paper is organized as follows. Section 2 explains the motivations for this work. The functionality of the environment used to develop the tool, DMS, are summarized in Section 3, while Section 4 describes the architecture of the proposed tool. Data on the impact of preprocessor conditionals on different software systems are reported in Section 5. Section 6 summarizes related work, while conclusion and work-in-progress are presented in Section 7.

2. Preprocessor-Conditioned types

In this section we explain, using some examples, the problems concerning the use of preprocessor conditioned variables. Figure 1 shows an example of preprocessor-conditioned variable declaration, and its use inside an expression.

Depending on the configuration, the type of the variable **`buffer`** may be:

1. **`char:`** if the expression *(defined (ALPHA) || defined (ATARI)) && defined (pyr)* is true;

```
#if defined(ALPHA) || defined (ATARI)
#if defined(pyr)
char * buffer;
#endif
#elif defined(i386)
int buffer;
#endif;
...
*buffer = 2;
```

Figure 1. Example of a preprocessor-conditioned declaration

2. **`int`:** if the expression *!(defined (ALPHA) || defined (ATARI)) && defined (i386)* is true;
3. ***`Undefined`*:** in all other cases.

This may lead to two categories of problems:

- Inconsistent use: e.g, declared as **`int`** but used as pointer;
- Used but not declared: The system could be compiled in a configuration where the variable `buffer` is never defined but is actually used at a certain point, since the use has accidentally not been excluded by preprocessor condition similar to the one excluding the declaration.

For particular configurations, these problems are detected at compilation time. However, this solution is infeasible for systems in which the number of possible configurations is huge, and it is simply impractical to compile every configuration before release to customers. For example, the Linux 2.4.0 Kernel contains more than 7000 files, and runs on 10 different processors [1]. It has 400 preprocessor switches, each of which may assume three different values (Y to include the code into the compiled kernel; N, or commented switch, to exclude the code, or M to produce a dynamically loadable module) drive the actual kernel configuration. This enables some $10*400^3$ possible different configurations (not all possible combinations of switches make sense, e.g., it is unlikely for a machine having several sound boards installed), which should be considered when testing the system.

Furthermore, there may be some problems that are discovered only during execution, when the affected portion of code is reached.

Most conditionals are inserted to compile the software system under different possible configuration. However, some are not and can effectively be ignored for type analysis. A typical (and, often, the most relevant) example is the preprocessor code used to avoid circular inclusion:

```
#ifndef MYFILE_H
#define MYFILE_H
…
#endif
```

Heuristics (often pattern-matching oriented) are adopted to avoid considering these preprocessor directives as conditions on variables.

3. DMS

The Design Maintenance System (DMS) [2] is a reengineering vision and toolkit that enables the analysis, translation, and/or reverse engineering of software systems. The DMS vision encompasses software development from initial synthesis, through subsequent enhancement and on to maintenance. The DMS vision incorporates several key ideas:

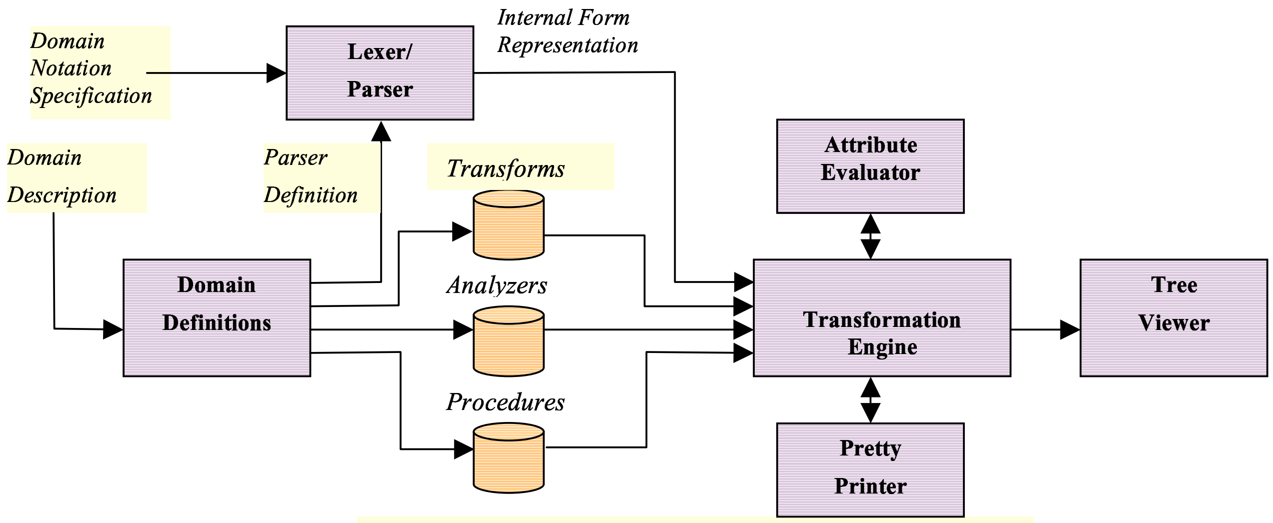


Figure 2. DMS Reengineering Toolkit Components

- *Design centric.* The DMS vision holds that the production and maintenance of the design is a key aspect in software maintenance and evolution.

- *Semantics based.* In the DMS view there is a correspondence between the designer's analytical efforts and the semantic capabilities provided by DMS.

- *Multilingual.* DMS provides facilities for a variety of languages. Each formal language is referred as a domain that includes a description of language's syntax, and as much (or as little) of the language's semantics as is necessary for the task at hand;

- *Industrial Scale:* DMS does not limit the size of the application that can be manipulated.

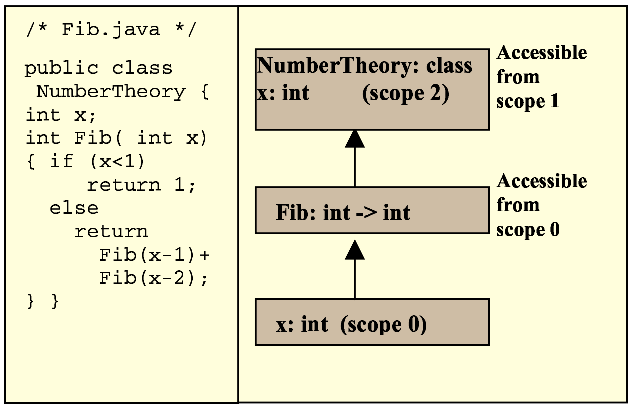
The DMS toolkit has been used for a number of commercially interesting tasks [3, 4], such as generation of domain tools, automated detection of duplicate code clones, code generation of factory controller programs from factory process specifications, implementation of code test coverage and so on.
A number of reengineering technologies are integrated into DMS. The main components of DMS Reengineering Toolkit, depicted in Figure 2, are:

- Lexer;
- Parser;
- Attribute evaluator;
- PrettyPrinter;
- Rule Applier; and
- API to manage AST and Symbol Table.

In particular, DMS provides an attribute grammar definition system arbitrary analyses. The DMS *Attribute Grammar* domain enables a domain engineer to specify attribute computations over grammar productions and terminals to compute values over syntax trees (i.e., parse trees). Such computations are commonly used to create symbol tables, resolve names, i.e. map name uses to name definitions, perform type checking and inferencing, as well as to perform other analyses to answer questions about domain instances often with the goal to manipulate domain instances. The main tool in the domain is the *AttributeEvaluatorGenerator* which, given an attribute grammar, produces a parallel program that performs the specified attribute computation.

DMS also provides a generalized symbol table management mechanism, capable of capturing a wide variety of scoping rules and type information. For fast access to information related to identifiers in a source program or specification it is necessary to record these identifiers together with collected information about various of their attributes, such as their scope of validity and their types. This information can be used to check the well formedness of the program, e.g. its type correctness, as well as during transformation of the source. The symbol table data type provides an easy way to record such identifiers and store or retrieve information associated to them. It supports the definition of hierarchical maps from identifiers to values together with visibility restrictions for those identifiers. An example of symbol table for a *fib* program is depicted in Figure 3.

Finally, DMS also provides a pattern and rule specification language, allowing the specification of conditional source-to-source rewrite rules on trees. The DMS rule specification language provides basic primitives to build conditions, patterns, rules, and rule sets using the surface syntax defined by string based domains as well as tree based compositions.



```
/* Fib.java */
public class
 NumberTheory {
int x;
int Fib( int x)
{ if (x<1)
     return 1;
  else
    return
      Fib(x-1)+
      Fib(x-2);
} }
```

(a) **(b)**

Figure 3. Example of DMS symbol table

The patterns and rules can have conditionals associated to them, describing restrictions on when a pattern actually matches a syntax tree or a rule is actually applicable on a syntax tree.

4. Architecture of Type-Checking tool

The architecture of the proposed tool is shown in Figure 4. Once the source code of the system to analyze has been parsed and the Abstract Syntax Tree (AST) produced, the *Conditional Name Resolver* builds an *Enhanced Symbol Table*, in which the type of each symbol is conditioned by an expression (see details in Section 4.1). Type-checking on the expressions contained in the AST of the software system is then performed for all the possible configurations.

One of the main advantages of the proposed tool is that, while with a traditional compiler can only compile and

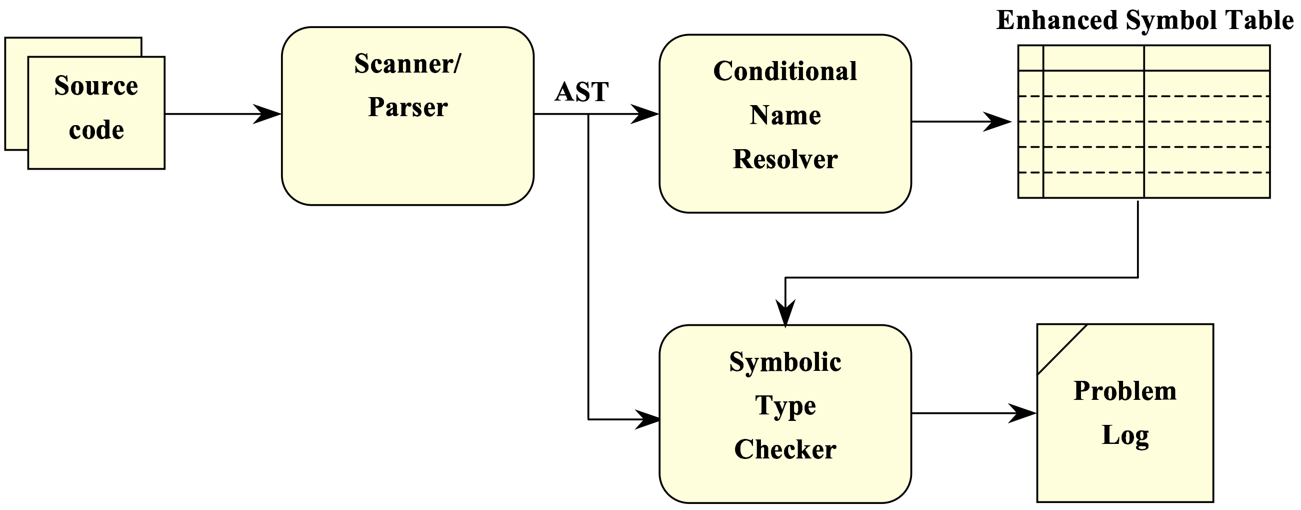


Figure 4. Architecture of the tool

check the software system in only one possible configuration at a time, this tool can type-check each expression for all possible combination of types variables involved in the expression. Finally a report log, containing the list of all possible inconsistencies, is produced.

4.1. Building the Enhanced Symbol Table

As described in Section 3, DMS builds the symbol table of a source code file using an attribute evaluator. However, the standard DMS *Name Resolver* for C does not take care of the preprocessor directives, unconditionally inserting symbols into the symbol table, or leaving to a preprocessor the task of expanding directives.

Symbol	Scope	Type	Conditional Expression
buffer	0	char*	(defined (ALPHA) || defined(ATARI)) && defined (pyr)
buffer	0	int	!(defined(ALPHA) || defined(ATARI)) && defined (i386)
…	…	…	…
x	1	float	True
k	2	int	defined(SPARC)

Table 1. An example of enhanced symbol table

What we aim to do is to associate, to each triple *symbol-scope-type*, a Boolean expression like those in the examples of Section 2. Moreover, it is worth noting that, in cases like Figure 1, the symbol table can contain several instances of the same couple *symbol-scope*, where the type varies with the conditioning expression (see Table 1). For a declaration outside the preprocessor conditionals, like variable `x` in the table, the Conditional Expression field is, obviously, set to `true`.

Starting from the original DMS *Name Resolver*, we added a new attribute *expr* to the attribute evaluator. This attribute brings the Boolean condition from the conditional directive to the variable declaration. More Boolean expressions are composed in presence of nested preprocessor conditionals.

At the root node of the AST the attribute *expr* is set to *true* (i.e., if a symbol is declared outside a preprocessor conditional, it always assumes the associated type, as in the original DMS symbol table). Appendix A shows how the new attribute flows through the grammar terms.

A preprocessor-conditioned code block can be described by one of the following grammar rules:

```
1. block = if_group block '#endif'
2. block = if_group block
           #else' block '#endif'
3. block = if_group block
           elif_group block
           '#endif'
4. block = if_group block
           elif_group block
           '#else' block
           '#endif'
```

where an *if_group* may be a directive:

```
5. if_group = #if expr
6. if_group = #ifdef IDENTIFIER
7. if_group = #ifndef IDENTIFIER
```

Let us consider now, as highlighted in Appendix A, the second case (for the other rules the behavior is similar).

The *if_group* passes the symbolic Boolean expression (*expr*) to the conditioned blocks, i.e., to both block grammar symbols in the right-side of the grammar rule (the expression is negated when passed to the second block, being the *else* branch of the condition).

The symbolic expression coming from the *if_group* is then combined with the expression coming form the outer block (i.e., from the left side of the grammar rule) by an AND operator, building a conditional expression for declarations inside nested preprocessor conditionals. Finally, the conditional expression is then simplified algebraically using rewrites. Further details about how expressions are composed and simplified are shown in Section 4.2.

```
public pattern
not_expression(e1:simp_constant_expression
): simp_constant_expression  =
      "!(\e1\:simp_expression)".


public pattern
and_expressions(e1:simp_constant_expressio
n, e2:simp_constant_expression):
simp_logical_and_expression
       = "(\e1\:simp_expression) &&
          (\e2\:simp_expression)".
```

Figure 5. Transformation rules for composing expressions

Once the attribute *expr* reaches a symbol declaration, the insertion of the symbol in the symbol table follows the schema explained by the pseudo-code shown in Appendix A: the function **AddCondSymbol** inserts all symbol information (name, scope, type), along with the conditional expression in the symbol table. If the declaration is not inside a preprocessor conditional, then the conditional expression assumes the Boolean value *true* (e.g., variable `x` in Table 1).

It is worth noting that, if the same symbol is already present in the same scope, then the function **AddCondSymbol** simply adds another instance of the couple type-conditional expression (e.g., the variable `buffer` in Table 1).

4.2. Composing and simplifying expressions

The DMS Rule Specification Language enables the specification of patterns and rules to compose and transform ASTs.

For composing expressions, two transformation rules were used:

1. To negate an expression, before passing it to the *else* branch of a conditional, and
2. To compose nested conditions using the Boolean AND operator.

The DMS transformation rules are shown in Figure 5.

The first rule defines a pattern, named *not_expression*. The expression in parentheses indicates that the AST node to be replaced must be of type *simp_constant_expression*; the type of the instantiated AST node is indicated after the colon. The expression in the right side indicates that the matched pattern *(e1)* will be transformed to *!(e1)*. A new AST node will be instantiated, whose type (*simp_constant_expression*) is indicated in the left side of

```
private rule eliminate_parentheses_27 (e:simp_expression):
    simp_expression->simp_expression
    ="(\e)"->"\e\:simp_expression".


private rule simplify_not_not(e:simp_unary_expression):
    simp_unary_expression->simp_unary_expression
    ="!!\e" -> e.


private rule
simplify_or_repetition_2(e1:simp_logical_and_expression,e2:simp_logical_and_expression):
    simp_logical_or_expression->simp_logical_or_expression
    ="\e1 || \e2 || \e2"->"\e1 || \e2".


private rule simplify_or_and_1a(e1:simp_logical_and_expression,
                                e2:simp_logical_and_expression):
    simp_logical_or_expression->simp_unary_expression
    ="\e1 || \e1 && \e2\:simp_inclusive_or_expression"->"\e1\:simp_unary_expression".
```

Figure 6. Examples of rules

the expression after the colon. Similarly, the second rule composes two AST nodes creating a *simp_logical_and_expression* node.

A further task performed by transformation rules is to simplify conditional expressions. Suppose we have a piece of code written as follows:

```
#if defined(i386)
#if defined(i386) || defined (intel)
int x;
#endif
#endif
```

The condition on the variable `x` should ideally be:

```
#if defined(i386)
```

Simplifications are performed by a set of transformation rules, applying well-known properties of the Boolean algebra, and pruning superfluous parentheses. Some examples of rules are reported in Figure 6. Further details can be found in [3]. It is worth noting that these rules work properly given that associative and commutative properties have been properly defined in the grammar for all Boolean operators.

4.3. Type-checking

In this subsection we will consider how we could extend our approach with a type-checking mechanism.

A mechanism for type-checking is similar to the mechanism for evaluation. In evaluation, an expression is processed to arrive at a value, while an expression is processed to determine a type for type-checking.

Indeed, a type-checker verifies that the type of a construct matches that expected by its context. For example, a type-checker verifies that a de-referencing operator is applied only to a pointer, or that indexing is done only on an array.

Type-checking is typically achieved by providing a mechanism to define type-constraints indicating the type of a variable [5].

The design and the implementation of a type-checker for a specific language is based on information about the syntactic constructs in the language, the notion of the types, and the rules for assigning types to language constructs. A collection of rules, for assigning type expressions to the various part of a program, is required.

The type of a language construct is referred in literature as "type expression". A type expression can be either a basic type or formed by applying a type constructor. The set of basic types and constructors depends on the specific language. For the C language all this information can be extracted from the C reference manual [6].

A convenient way to represent and evaluate a type expression is to use a graph. It is possible to construct a tree for a type expression, with inner nodes for type constructors and leaves for basic type, type name, and type variables.

In our tool the traditional approach to implement a type-checker will be extended to consider also preprocessor conditioned variables declarations and their use inside expressions. We aim to address also particular situations in which variables are never defined in some configurations, but also used in the program construct because they erroneously have not been considered by preprocessor condition, or cases in which variables are declared with a specific type for specific configuration but used as a different type (i.e., a language construct has been erroneously reached).

4.4. Examples

In order to better understand what kind of faults we propose to discover, this section reports some examples of failed type-checking.

Consider, for example, the following C code:

```
{ #if ARRAY
      char **x;
  #else
      char *x;
  #endif
  ......
  x="Hello world!";
}
```

It is clear that, in the configuration *ARRAY,* `x` is a pointer-to-pointer to `char`, therefore the assignment will result as faulty.

Another example is the following:

```
{ #if M
      byte *address;
  #endif
  #if N
      char *address;
  #endif
      ......
      printf("the address is:%s", address);
      ......
}
```

In this case the `printf` statement works correctly only in the *M* configuration.

5. Impact of preprocessor conditionals

In this section the impact of preprocessor conditionals on variable declaration is analyzed and discussed, demonstrating the potential usefulness of the approach. Seventeen open source software systems were analyzed, most of which also studied in [7]. We added two systems, `mozilla` and the `Linux kernel 2.4.18` for which, in our opinion, preprocessor conditionals play a fundamental role. We found that most of the configuration-dependent declarations were contained in the header files, and so we concentrated our analysis there. As discussed in Section 2, we avoided considering as conditions preprocessor directives used to prevent circular inclusion.

Figure 7 reports the percentage of preprocessor conditioned declaration in the software systems analyzed. In most cases, the percentage is not negligible, and is around 20% (i.e., `m4`, `gzip`, `gnuplot`, `ghostview`, `gawk`, `emacs`, `bison`). There are cases, which should be analyzed in more closely where the percentage is close to 40% (e.g., `perl`, `mozilla`, `gs`, `gnuplot` and the `Linux Kernel`),

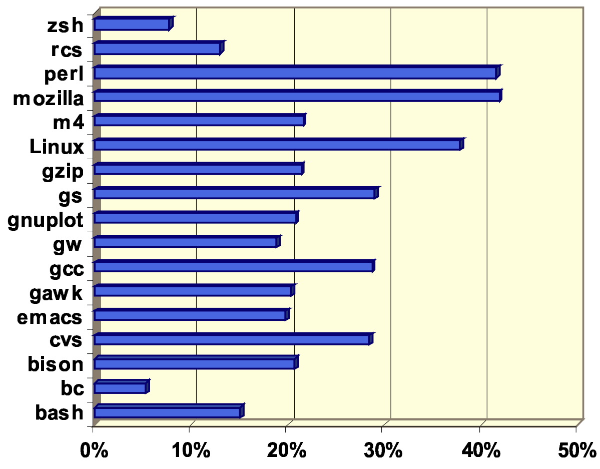


Figure 7 – Percentage of preprocessor-conditioned declarations

High percentages for `mozilla` (some examples are shown in Figure 9) are due to factors such as:

- Handling platform-dependent code; and
- Verifying if the configuration under which the system will be compiled includes certain libraries and components.

As highlighted in Figure 9, complex and nested conditions should be handled: this enforces the need for the expression simplifier explained in Section 4.2.

Similarly, analyzing `perl`, we experienced that preprocessor conditionals are targeted to compile the interpreter on different platforms and to check if some `perl` add-ons and libraries are included in the configuration to be built.

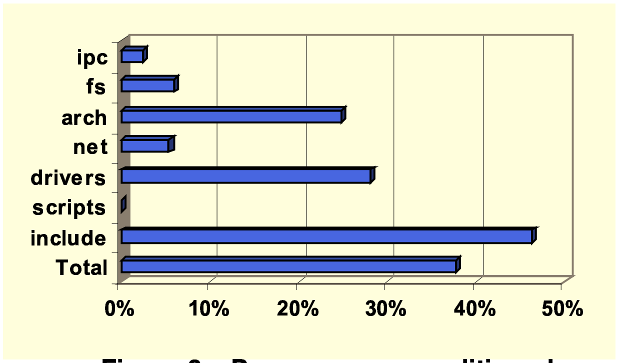


Figure 8 – Preprocessor-conditioned declarations in the Linux Kernel subsystems

A more focused analysis was performed on the Linux Kernel. The objective was to analyze the impact of preprocessor conditioned declarations on the different Kernel subsystems.

To perform an analysis on different subsystems, we followed the first-level depth subdirectory decomposition also used for clone-detection in [1]. In other words, we analyzed separately general header files contained in the `include` directory, and then those of the six major subsystems (`ipc`, `fs`, `arch`, `net`, `drivers`, `scripts`). Results are reported in Figure 8, together with the overall results (the *Total* bar) from Figure 7.

The percentage in the include directory was significant, due to the fact that most of the *Linux Kernel* code is customized

```
#ifdef XP_WIN
#ifdef XP_WIN16
#if defined (XP_WIN) || defined (XP_OS2)
#ifdef XP_MAC
#ifdef XP_UNIX
#if !defined(XP_RANDOM) ||
    !defined(XP_SRANDOM)
#if defined(UNIXWARE) ||
    defined(_INCLUDE_HPUX_SOURCE) ||
   (defined(__sun) && defined(__svr4__))
|| defined(SNI) || defined(NCR)
                    …
```

Figure 9. preprocessor conditionals in `mozilla` code

during pre-compiling configuration (i.e., establishing the architecture, what drivers to include, etc.). As expected, hardware-specific code (`drivers` and `arch` subsystems) exhibit a considerable percentage, especially more independent subsystems, such as `net`, `fs` and `ipc`.

6. Related work

Programmers tend to consider it necessary to port C software from one configuration to another, as C runs on "practically everything".

When differences among systems cause difficulties, the usual first solution adopted by programmers is to write two different versions of the code, one per system, and use `#ifdef` to choose the appropriate one. However, the large use of `#ifdef` to attempt at portability is usually a source of several problems. The result is usually an unreadable, and difficult to maintain software system [8].

Ernst *et al.* [7] presented the first empirical study of the use of the C macro preprocessor. They analyzed 26 packages, to determine the practical use of preprocessor directives. The authors also proposed a taxonomy of various aspects of preprocessor use. This paper reported data regarding the prevalence of preprocessor directives, macro body categorizations, use of the C preprocessor to achieve features impossible in the underlying language, inconsistencies and errors in macro definitions and uses, and dependences of code upon macros.

Hu *et al.* [9] coped with conditional compilation problems. They presented an approach based on symbolic execution of preprocessing directives. Their goal was to find the simplest sufficient condition to reach/compile a line of code containing a preprocessor directive, and the full condition to reach/compile it. Experiments have been conducted on Linux Kernel, using a tool that automates the approach presented.

Livadas and Small [10] addressed problems concerning source code containing preprocessor constructs, such as included files, conditional compilation, and macros. The authors proposed a mapping from token in the preprocessor output to the source file(s), and discussed the use of these correspondences, through an internal program representation, for maintenance purpose jointly with techniques including program slicing, ripple analysis, and dicing.

Baxter and Mehlich [3] explained disadvantages in having preprocessor conditionals inside code when the presence of configuration dependent code loses its utility, and proposed a method for its removal. A method to simplify preprocessor Boolean expressions was adopted to easily evaluate expressions.

Type inferencing is also present in other code-analysis tools. For example, TXL [11] developers implemented a type inferencing and checking prototype as an attribute grammar.

7. Conclusions

This paper presented an approach and the architecture of a tool devoted to the automatic detection of faults caused by wrong use of configuration-dependent variables.

This was obtained building an *enhanced symbol table*, in which the type of a variable depends on a Boolean expression obtained composing and then simplifying preprocessor conditionals dominating the variable itself.

This will allow a more effective type-checking with respect to a traditional compiler, since the latter needs to compile the system for each individual configuration to be tested, while the former can automatically check expressions present in the code for all the possible configurations.

Possible impact of the tool was analyzed computing the percentage of preprocessor conditioned variables present in 17 different software systems, and analyzing the purpose of these conditionals.

Work in progress is devoted to complete the tool (at the moment only the enhanced symbol table builder is implemented), and to apply it on some open source systems, like those preliminarilily analyzed in Section 5.

8. Acknowledgments

We would like to thank Gerardo Canfora and Giulio Antoniol for their stimulating discussions and valuable advice. Lerina Aversano and Massimiliano Di Penta were partially supported by the ASI grant I/R/091/00.

References

[1] G. Casazza, G. Antoniol, U. Villano, E. Merlo and M. Di Penta. Identifying clones in the Linux Kernel. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation*, Florence (Italy), November, 2001, pp. 90-97.

[2] I. D. Baxter. Design Maintenance Systems. C*ommunications of the ACM* 35(4), April, 1992, Association for Computing Machinery, New York.

[3] I. D. Baxter and M. Mehlich. Semantic Designs Preprocessor Conditional Removal by Simple Partial Evaluation. *Workshop on Analysis, Slicing, and Transformation*,2001,Germany.

[4] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. *In*

Proceedings of the International Conference on Software Maintenance, pages 368–377, Bethesda, Maryland, USA, November 1998.

[5] A.V. Aho and J.D. Ullman. Principles of Compiler Design, Addison Wesley 1977.

[6] C Language ISO/IEC Standard 9899

[7] M. D. Ernst, G. J. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. To appear in *ACM* Transactions on Software Engineering and Methodology.

[8] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In USENIX, Summer 1992 Technical Conference, San Antonio (Texas), June, 1992, pp. 185-197.

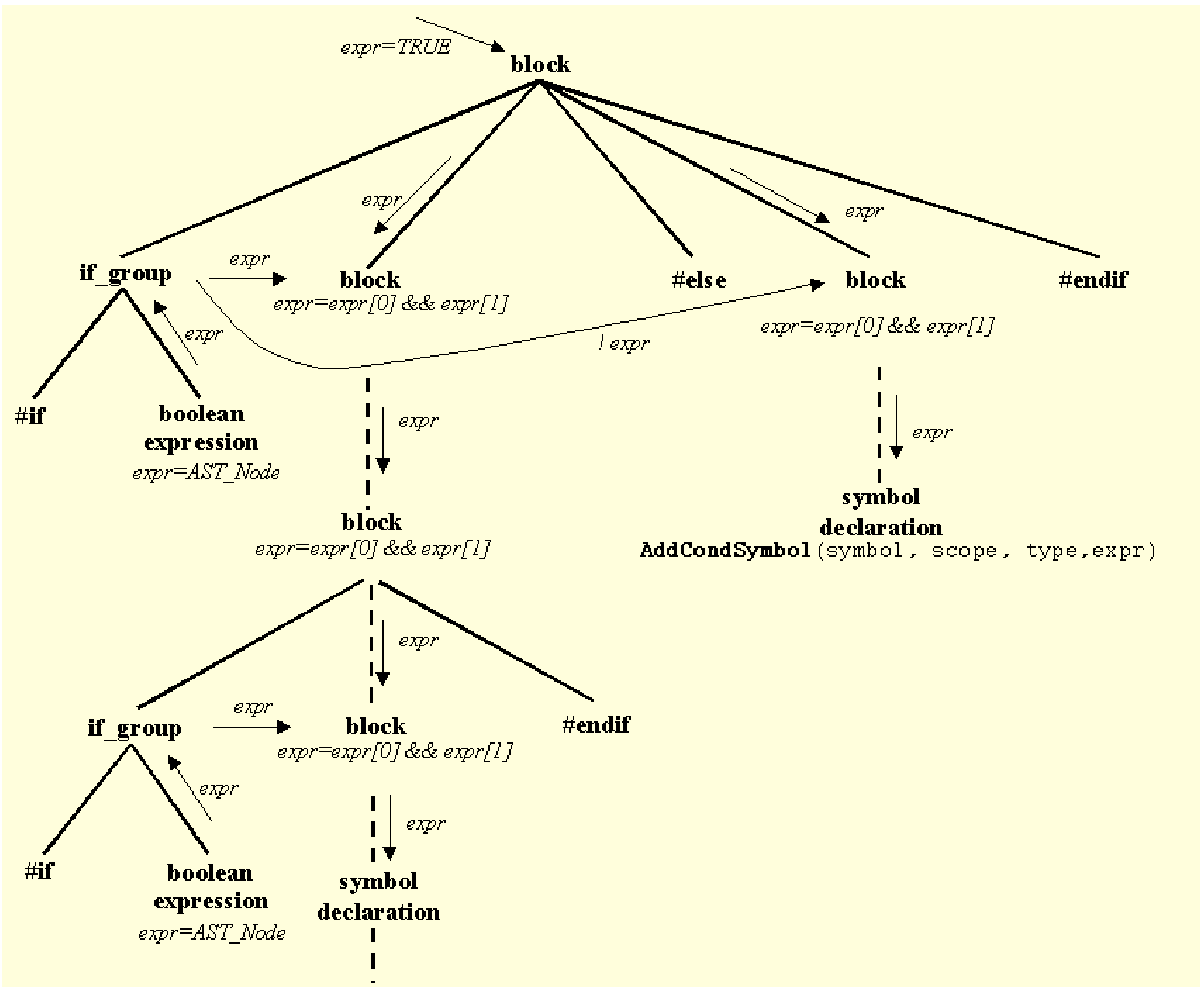
[9] Y. Hu, E. Merlo, M. Dagenais and B. Lague. C/C++ Conditional Compilation Analysis Using Symbolic Execution, In *Proceedings of the International Conference on Software Maintenance*, San Jose, California, USA, 2000, pp. 196-206.

[10] P. E. Livadas and D. T. Small. Understanding code containing preprocessor constructs. In *Proceedings of the Third Workshop on Program Comprehension*, Washington, DC, USA, November, 1994, pp. 89-97.

[11] TXL Home Page (http://www.txl.ca).

APPENDIX – A

Graphical representation of the attribute flows through the grammar terms.



expr=TRUE
block
expr
expr
if_group
expr
block
expr=expr[0] && expr[1]
#else
block
#endif
expr
! expr
expr=expr[0] && expr[1]
#if
boolean expression
expr=AST_Node
expr
expr
block
expr=expr[0] && expr[1]
symbol declaration
AddCondSymbol(symbol, scope, type,expr)
expr
if_group
expr
block
expr=expr[0] && expr[1]
#endif
expr
expr
#if
boolean expression
expr=AST_Node
symbol declaration