Extracting variability from C and lifting it to mbeddr

Federico Tomassetti
Politecnico di Torino
C.so Duca degli Abruzzi 24,
Torino, Italy
federico.tomassetti@polito.it

Daniel Ratiu
Fortiss
Guerickestr. 25,
Munich, Germany
ratiu@fortiss.org

***Abstract*—Information about variability is expressed in C through the usage of preprocessor directives which interact in multiple ways with proper C code, leading to systems difficult to understand and analyze. Lifting the variability information into a DSL to explicitly capture the features, relations among them and to the code, would substantially improve today's state of practice. In this paper we present a study which we performed on 5 large projects (including the Linux kernel) and almost 30M lines of code on extracting variability information from C files. Our main result is that by using simple heuristics, it is possible to interpret a large portion of the variability information present in large systems. Furthermore, we show how we extracted variability information from ChibiOS, a real-time OS available on 14 different core architectures, and how we lifted that information in mbeddr, a DSL-based technology stack for embedded programing with explicit support for variability.**

***Keywords*-software product lines; abstraction lifting; embedded software; projectional editors; reverse engineering.**

I. Introduction

For decades the C language has been the language of choice in developing embedded systems[1]. Nevertheless development in C is affected by problems due to the presence of preprocessor directives intermingled inside the C code. The preprocessor complement fundamentally to the expressiveness of the C language permitting conditional compilation, constant definitions, or basic meta-programming support. However the preprocessor favors also the presence of bugs [1] and lead to code that is extremely difficult to understand and analyze. For example, the `ifdef` directive permits to implement variability with the goal of obtaining portability or implementing product lines but it can be easily abused leading to situations in which the code is hard to understand and all possible variants are extremely difficult to analyze [2].

Mbeddr[2] [3] is a technology stack based on language engineering that defines domain specific extensions of C for embedded programing – e.g., components, state-machines, physical units. Mbeddr is built on top of the Meta Programming System (MPS) from JetBrains, which is a projectional language workbench. Since the extensions are based on C, mbeddr can be easily integrated with existing C code. While most of the syntax and semantics of C is preserved in mbeddr some notable features were removed to create a new language easily analyzable. Notably preprocessor directives are not supported and for many common usages of the preprocessor, explicit language constructs are provided.

```
void foo()
{
#ifdef FEAT_A && !FEAT_B
    call_a();
#elif defined(VERS) && VERS > 10
    // do nothing
#else
#ifdef FEAT_B
    call_b();
#else
    log("Unsupported operation");
#endif
#endif
}
```

Figure 1. Variability in C is expressed implicitly through the preprocessor.

In mbeddr is possible to represent explicitly software product line concepts [4] with consequent advantages in terms of analizability (e.g., if feature models are consistent) [5] and comprehension. To operate with the existing code base written in C, an importer is needed to first capture variability expressed using preprocessor directives (like in Figure 1) and then to re-express it using the specific constructs provided by mbeddr (like in Figure 2). As shown in Figure 2, in mbeddr, feature models and configurations are captured through domain specific constructs and the linking of features to code is explicitly represented by annotations on the statements. The annotations contain an expression with references to features which determine if the statement will be included under a particular configuration. These expressions are called presence conditions. Due to the projectional editing capabilities of MPS, programs can be displayed as the entire product line or as individual variants. It means the developer could visualize all possible statements with their presence conditions, or only the statements that will be included when a particular configuration it is used. Importing the C code and variability in mbeddr means to lift the level of abstraction from token level (the level at which the preprocessor operates) to a domain specific level where variability concepts are expressed as first class citizens.

[1]According to the "Transparent Language Popularity Index" it is still the most popular programming language in general in January 2013.

[2]http://mbeddr.com

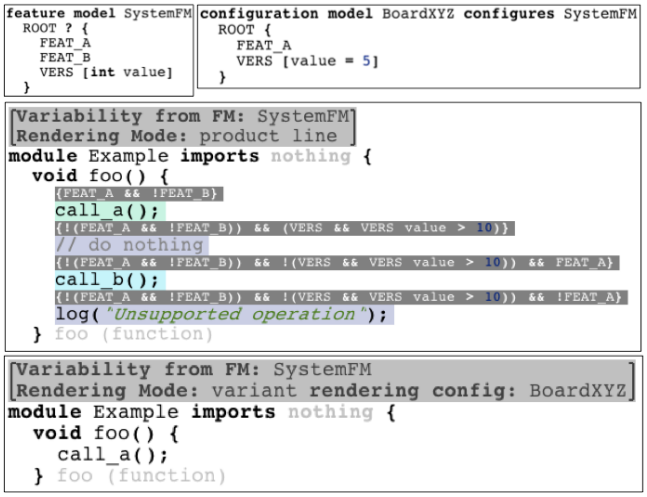

Figure 2. Variability in mbeddr: on top you see the feature model (left) and a configuration of that feature model (right). Below there are two projections of the code: the first one shows all the possible variant, the second one the variant corresponding to the configuration.

The rest of the paper is organized as follows: we start describing how variability is represented in C (Sect. II) and later present an empirical analysis of usages of variability among large C projects (Sect. III). In Sect. IV we report about our experience in extracting variability from ChibiOS source code to mbeddr. Finally we introduce related work (Sect. V) and draw our conclusions (Sect. VI).

II. How variability is expressed in C

The preprocessor when it is invoked can receive a set of parameters, called the *initial configuration*. The *initial configuration* specifies which macros are initially defined and their initial values. During the preprocessing new macros can be defined, and the existing ones can change their value or being undefined. This process is called *configuration processing*. Based on the current configuration (the set of macros being defined and their associated value at a given moment) declarations and statements are included or excluded in the code to be compiled. The expressions determining the inclusion/exclusion of C elements are called *presence conditions*. In addition to that, the preprocessor statements `#error` and `#warning` can be used to issue errors and warnings to the user when a particular configuration is not acceptable or it is deprecated.

Some approaches to extract variability from C (e.g., [6]) require to process the initial configuration and rewrite presence conditions in term of the initial configuration instead of the current configuration. While this is a sound solution for analysis, we conjecture that in C important information is expressed by the combinations of these two different mechanisms: *configuration processing* and *presence conditions*. The first can be used to specify derivation rules that determine the values of symbols later used in *presence conditions*. Consider the example presented in Listing 1. First the *configuration processing* determines that `X_SHOULD_BE_ACTIVATED` will be defined only when the condition `defined(FEATURE_A) && (PARAM_B >0x1010` will have the value `true`. Subsequently, based on the fact that `X_SHOULD_BE_ACTIVATED` is defined or not, two different statements could be included or excluded (both of them located inside the function `foo`). While we could rewrite the presence conditions to `defined(FEATURE_A) && (PARAM_B>0x1010)` and discard all the *configuration processing* we think that this would cause a loss of information which could be useful while maintaining the system.

```
#if defined(FEATURE_A) && (PARAM_B>0x1010)
#define X_SHOULD_BE_ACTIVATED
#endif
void foo()
{
#ifdef X_SHOULD_BE_ACTIVATED
  invoke_x_init();
#endif
  ...
#ifdef X_SHOULD_BE_ACTIVATED
  invoke_x_release();
#endif
}
```

Listing 1. Example of C and preprocessor code containing both configuration processing and presence conditions

III. Empirical analysis of variability in large C projects

To lift the variability information in mbeddr, we need to understand how is it expressed in large C programs, this is the goal of the analysis introduced in this section. In particular we want to investigate if preprocessor directives in the context of large projects are used in a disciplined way to represent variability. If that is the case we could identify usage patterns of the preprocessor that cover the majority of cases in the practice and exploit them in interpreting variability.

A. Research questions

Specifically, we aim to answer the following questions:

RQ1) *Which are the typical building blocks in presence conditions?* This is important in order to understand which kind of expressions we need to support in the higher level configuration language.

RQ2) *Which changes (re-* #define*s and* #undef*s) are operated on a defined symbol?* Depending on changes upon defined symbols, defines can be lifted (or not) as constant configuration values.

RQ3) *Are* #error *and* #warning *used in practice?* If they are, it could be possible to extract feature model constraints from them.

B. Analysis approach

In this section we present the general approach we adopted to answer our research questions. We present the projects we chose to analyze (III-B1), which information we extracted

from source files and how (III-B2), how we modelled variation points (III-B3).

1) Projects: To perform our analysis we selected established large projects from different domains. They are:

- **Apache OpenOffice:** it is a suite of six personal productivity applications. It is ported on Windows, Solaris, Linux, Macintosh and FreeBSD. It derives from StarOffice, which was developed since 1984.
- **Linux:** Linux is an OS kernel developed since 1991. It is arguably one of the existing projects which is more portable being available on more than 20 architectures.
- **Mozilla:** Mozilla is a suite of different projects including the Firefox browser for desktop and mobile systems and the Thunderbird e-mail client. It was created by Netscape in 1998.
- **Quake:** it is a videogame released during 1996. It runs on DOS, Macintosh, Sega Saturn, Nintendo 64, Amiga OS.
- **VideoLAN:** It is a multimedia player, supporting a large variety of audio and video formats. It has been ported to Microsoft Windows, Mac OS X, Linux, BeOS, Syllable, BSD, MorphOS, Solaris and Sharp Zaurus. The first release is dated 2001.

Some data about the dimension of projects chosen is reported in Table I. We chose these projects because they are multi-platform projects, from different domains and they are written mainly in C or other languages sharing the same preprocessor. In particular Apache OpenOffice, Mozilla and VideoLAN contain also Objective-C and C++ files. We considered more than 73.000 files with a total of more than 2.1 millions of preprocessor statements.

2) Information extraction: We focus on the `define` statements because they are used to implement *configuration processing* and on the `ifdef`, `ifndef`, `if`, `elif` and `endif` statements because they can be used to express *presence conditions*. We excluded from our analysis statements that, while being of one of the previous types, were not used to express variability. To do that we built for each file a model of the information that are relevant to the preprocessor. We call this model the *preprocessor model*.

Preprocessor model: A *preprocessor model* is an ordered list of the elements contained in a source file. Possible elements are: preprocessor statements, blank lines, comment lines[3] and code lines. Preprocessor statements are recognized from lines not contained in comments which starts with the '#' symbol. They can span across multiple lines when a line is terminated with the '\' character. Comment lines are lines containing whitespaces and comments, blank lines are lines composed only by whitespaces and not included in a multi-line comment. Finally code lines are lines containing some code (they can also include comments). Note that while not parsing the C/C++/Objective-C code the parser have still to be able to handle correctly string and char literals to recognize comments. Data about the number of lines contained in each project are reported in Table I.

Parsing preprocessor expressions: In addition to classify the lines, we parsed the expressions contained in preprocessor statements and inserted them in the *preprocessor model*. In particular we calculated the value of the conditions associated to preprocessor statements `ifdef`, `ifndef`, `if` and `elif` and the expressions specified by `define` statements. While statements `ifdef`, `ifndef` can express only simple presence conditions (based on the presence or absence of one single configuration symbol), `if` and `elif` can specify very complex expressions. To parse those complex expressions we used the grammar presented in Listing 1 (whitespaces and comments were ignored, including the backslash followed by a newline, which is used inside preprocessort statements to continue on the next line). Our parser implementation takes in account the precedence between operators.

⟨*expression*⟩ ::= '(' ⟨*expression*⟩ ')'
| ⟨*define*⟩
| ⟨*flag_value*⟩
| ⟨*logical_binary_op*⟩
| ⟨*comparison_op*⟩
| ⟨*number_literal*⟩
| ⟨*char_literal*⟩
| ⟨*math_op*⟩
| ⟨*bitwise_binary_op*⟩
| ⟨*logical_not*⟩
| ⟨*bitwise_not*⟩
| ⟨*macro_function_call*⟩
⟨*define*⟩ ::= 'defined' '(' ⟨*identifier*⟩ ')'
| 'defined' ⟨*identifier*⟩
⟨*flag_value*⟩ ::= ⟨*identifier*⟩
⟨*identifier*⟩ ::= +[_a-zA-Z][_a-zA-Z0-9]*
⟨*logical_binary_op*⟩ ::= ⟨*expression*⟩ ('&&'|'||') ⟨*expression*⟩
⟨*comparison_expression*⟩ ::= ⟨*expression*⟩ ('<='|'>='|'>'|'<'|'=='|'!=') ⟨*expression*⟩
⟨*number_literal*⟩ ::= ...
⟨*char_literal*⟩ ::= ...
⟨*bitwise_binary_op*⟩ ::= ⟨*expression*⟩ ('«'|'»'|'&'|'|') ⟨*expression*⟩
⟨*math_op*⟩ ::= ⟨*expression*⟩ ('+'|'-'|'*'|'/'|'%'|'^') ⟨*expression*⟩
⟨*logical_not*⟩ ::= '!' ⟨*expression*⟩
⟨*bitwise_not*⟩ ::= '~' ⟨*expression*⟩
⟨*macro_function_call*⟩ ::= ⟨*identifier*⟩ '(' (⟨*expression*⟩ (',' ⟨*expression*⟩)*)? ')'

Grammar 1. Grammar used to parse presence conditions. Definitions of literals are omitted.

Using this grammar we were able to parse correctly a large majority of the conditions expressed: out of more than 185K expressions analyzed we could not parse only three. Expressions that could not be parsed are reported in Listing 2. The same grammar can be used to parse a portion of the `define` statements, when valid expressions are assigned to symbols. In some cases however `define` can assign to symbols arbitrary tokens instead of expressions, for example complete or incomplete statements. This is not a problem relavant for feature model and configura-

[3]Comment lines were included in the model because as future work we aim to associate comments to preprocessor statements (based on adjacency) and import also them.

Description			Files					Lines				
Project	Domain	Version	C	C++	Obj-C	H	Total	Blank	Comm.	Code	PP	Total
AOO	Productivity	V. 3.4.1 Linux DEB	158	11,029	51	12,107	23,345	1.2M	1.4M	5.3M	430K	8.2M
Linux	OS	Vv. 3.6.5	17,448	0	0	14,379	31,827	2.0M	2.2M	9.0M	1.3M	14.5M
Mozilla	Web	Tag FIREFOX_AURORA_19_BASE	3,118	4,502	180	8,050	15,850	807K	991K	3.7M	326K	5.9M
Quake	Gaming	Commit bf4ac424ce...	240	0	0	145	385	34K	43K	123K	8K	198K
VideoLAN	Multimedia	V. 1.3.0	778	255	81	1,311	2,425	97K	110K	426K	45K	678K
Total			21,742	15,786	312	35,992	73,832	4.1M	4.7M	19M	2.1M	29.5M

Table I
SIZE OF THE PROJECTS CONSIDERED. AOO = APACHE OPENOFFICE, H = HEADER, COMM. = COMMENT, PP = PREPROCESSOR.

tion extraction because symbols with syntactic content are not referred inside *presence conditions*, which have to be evaluable expressions. Parsing the values of both object-like and function-like symbols but excluding `define` with empty values, expressions parsable by our grammar ranged from 82% (for VideoLAN) to 95% for Linux.

```
// Mozilla
#if defined(LARGEFILE64_SOURCE) && - \
   _LARGEFILE64_SOURCE - -1 == 1
#if -_LARGEFILE64_SOURCE - -1 == 1
// Apache OpenOffice
#if NFWORK < (NAM$C_MAXRSS + 1)
```

Listing 2. All the expressions not parsed by our grammar

Preprocessor usages excluded: We used the *preprocessor model* to identify the preprocessor statements that were not related to variability. We defined two patterns to be searched in the model, ignoring blank and comment lines. In particular we excluded:

- **Double inclusion guards protecting modules:** we recognized them when a `ifndef` (or an `if` with an expression of type `!defined(SYMBOL)`) was at the very beginning of the file, immediately followed by a `define`, which: i) defined the same symbol used by the previous statement, ii) had no value specified. Finally a `endif` had to be the last element of the file. When recognizing this pattern we ignored the three preprocessor statements involved. This pattern was recognized in most of header files. It is used to prevent an accidental double inclusion of the same header file.
- **Redefinition guards:** we recognized them when a `ifndef` (or an `if` with an expression of type `!defined(SYMBOL)`) was followed by a `define` which defined the same symbol. This line had to be followed by an `endif`. When recognized the first and the third statements were excluded, while the second was marked as a guarded redefinition. Redefinition guards are often used to avoid warnings from the compiler.

3) Calculate conditions of variation points: As we discussed in Section II `if`, `ifdef` and `ifndef` can be used to define presence conditions. These directives open constructs which are terminated by an `endif` and can contain one `else` clause and any number of `elif` clauses. Each of these constructs individuate one or more portions of codes, which can contain other preprocessor statements or C elements. It is possible to insert other conditional constructs inside these portions, i.e., it is possible to have annidated conditional constructs. The portions of code individuate by the constructs are classified in three kinds:

- **Then block:** this is the area between the the `if`, `ifdef` or `ifndef` opening the construct and the first among `elif`, `else` or the `endif` which are parts of the same construct (i.e., we do not consider `elif` or `else` or the `endif` of annidated constructs).
- **Else block:** this is the area between the `else` and the `endif` closing the construct.
- **Elif block:** this is the area between the `elif` and the next `elif`, the `else` or the `endif` block of the same construct.

We map each `if` / `ifdef` / `ifndef` construct to a *Variation point* and each of its block to a *Variation point block*.

For each *Variation point block* a *specificCondition* and a *complexiveCondition* can be calculated. The *specificCondition* of a *ThenBlock* is just obtained from the expression following the `if`, `ifdef` or `ifndef` statement. In the case of the *ElifBlock* it is composed through a logical *and*: i) the condition of the corresponding *ThenBlock* negated, ii) the condition of all the preceding *ElifBlocks* negated, iii) the condition created from the expression following the specific `elif`. In the case of *ElseBlock* it is created composing through a logical *and*: i) the condition of the corresponding *ThenBlock* negated, ii) the condition of all the *ElifBlocks* negated. The *complexiveCondition* corresponds to the *specificCondition* if the block is part of a *VariationPoint* which is not annidated, otherwise it corresponds to the *complexiveCondition* of the block containing the *VariationPoint* in logical *and* with the *specificCondition* of the block.

C. Results and discussion

In this subsection we present how we addressed each RQ and the corresponding results (III-C1, III-C2, III-C3). Later we discuss our findings (III-C4). All the analysis do not consider the statements excluded for the reasons explained in Par. III-B2.

1) Addressing RQ1: Presence conditions: To answer this question we analyzed all the expressions from `if`, `ifdef`, `ifndef` and `elif` statements. For each type of expression (*Identifier*, *NumberLiteral*, *ComparisonOp.*, etc.) we counted in how many of the expressions considered it was used. To do that we looked at the type of the expression itself and the type of all its sub-expressions, recursively.

Results: We report frequencies of the different types of expression in Table II. We can see that, as easily predictable, identifiers are referred in most of the presence conditions (84.6%-98.1%). Presence conditions which instead do not refer to identifiers or macro function calls are constants: they are always true or false, independently of the current configuration. Many of the expressions not referring to identifiers are composed by only one costant, either '0' (false) or '1' (true). An `if` having as expression '0' cause the exclusion of all the elements contained in the *ThenBlock*. An `if` having as expression '1' leaves always untouched the elements in the *ThenBlock*. The remaining expressions without identifiers could still have a documentation role, showing the reasoning process bringing to include or exclude a particular set of statements. The most common operations are logical operations (`!`, &&, ||) which are present in many presence conditions (between 1/5 to 2/3 of the expressions considered). Comparison operations (<,>,<=,>=,==,`!=` are also relevant as well as number literals. Math (+,-,∗,/,%,∧) and bitwise operations («,»,&,|,) are very infrequent (they appear in less than 1% of the presence conditions). Quite infrequent are also macro function calls which are not used at all in one of the projects considered and seem to be marginally relevant only in VideoLAN and Linux (being contained in slightly more than 1% of all examined *presence conditions*). Observing the nature of macro functions used in presence conditions we noticed that quite frequently they are just implemented using stringifications[4] to compose different tokens creating a new identifier.

2) Addressing RQ2: Configuration processing: Technically the value of a macro can vary during the execution of the preprocessing. A scenario like the one presented in Listing 3 is therefore possible. In this example two functions (`foo_a` and `foo_b`) have the same *presence condition* (`XYZ` have to be defined) but because of the changes in the definition of `XYZ` (initially defined and then undefined) `foo_a` will be included while `foo_b` will be not.

```
#define XYZ
#ifdef XYZ
void foo_a(){};
#endif
#undef XYZ
#ifdef XYZ
void foo_b(){};
#endif
```

Listing 3. Example of configuration processing varying the value of a macro

The designers of Mbedder considered these consequences of the configuration processing confusing and do not support it in their variant of C; they instead consider configuration values to be constant.

While in general preprocessing symbols can be used in very different ways, we examined how frequently they are used as simple constants. We found three cases in which they behave as simple constants. To individuate instances of these cases we analyzed how a particular symbol was defined, re-defined or undefined in the scope of a complete project (because the preprocessor do not implement local scopes). One condition applies to all these cases: the symbol considered should be never undefined, because it would mean to limit its scope, while we are looking for symbols behaving as constants which are available to the whole system. The cases considered are:

- **Symbols defined once:** symbols that are defined just once in the scope of the project considered.
- **Symbols re-defined always to the same value:** symbols that are defined two or more times but they are assigned always exactly the same expression (possibly the empty value, meaning that they are defined but they have not an associated value).
- **Symbols re-defined under different conditions:** symbols which are defined two or more times, but every time they are defined under a particular condition they are defined to the same value.

To be able to recognize the symbols re-defined under different conditions we had to be able to calculate the *presence condition* under which a particular definition would be used. Consider the example given in Listing 4. In that example the same symbol (`VAL`) could assume different values. The value 1 is assumed only when the condition `defined(A) && defined(B)` is satisfied, the value 2 is assumed when the condition `defined(C)` || `defined(D)` is satisfied, otherwise the symbol remains undefined. To calculate the *presence condition* of a given definition is not trivial because `if`, `ifdef` and `ifndef` constructs can be annidated and also the role of `elif` and `else` have to be considered. To this operation we used the technique presented in Sub-subsection III-B3.

```
#if defined(A) && defined(B)
#define VAL 1
#endif
#if defined(C) || defined(D)
#define VAL 2
#endif
```

Listing 4. Example of definitions under different presence conditions

Results: Table III reports some data about the number of definitions and undefinitions considered and the number of symbols involved. It is possible to notice that the number of symbols undefined is many times smaller than the number of symbols defined. A symbol can be undefined for different reasons. One reason is to avoid compiler warnings: a symbol already defined can be undefined immediately before a statement defining it again. Another reason is to mimic the concept of scope: by undefining the symbol we are guaranteed previous definitions preceding the undefinition will not affect the code following the undefinition.

[4]See http://gcc.gnu.org/onlinedocs/cpp/Stringification.html for an explanation of stringification.

Expr. Type	AOO	Linux	Mozilla	Quake	VideoLAN	Range
Identifier reference	98.1	95.8	97.1	84.6	93.4	84.6-98.1
Number literal	7.9	7.0	6.5	15.7	10.0	6.5-15.7
Logical op.	66.3	17.9	22.3	28.3	20.9	20.9-66.3
Comparison op.	6.3	3.4	4.0	0.3	3.4	0.3-6.3
Math op.	0.04	0.1	0.1	0	0.3	0-0.3
Bitwise op.	0.01	0.5	0.01	0	0.3	0-0.5
Macro function call	0.01	1.3	0.2	0	1.5	0-1.5
'0'	1.8	3.5	2.7	14.6	5.5	1.8-14.6
'1'	0.2	0.5	0.2	0.8	0.5	0.2-0.8

Table II

PERCENTAGE OF PRESENCE CONDITIONS CONTAINING THE GIVEN TYPE OF EXPRESSION ACROSS THE DIFFERENT PROJECTS CONSIDERED. FOR THE EXPRESSIONS 0 AND 1 IT IS INSTEAD THE NUMBER OF EXPRESSIONS CORRESPONDING EXACTLY TO 0 OR 1. LAST COLUMN REPORT THE RANGE (THE SPACE DETERMINED BY THE MINIMUM AND MAXIMUM VALUES AMONG ALL PROJECTS).

	Definitions						Undefinitions		Errors and warnings		
Project	Sym	Def	D1	D2	D3	D4+	Sym	Undef	Error	Warning	Perc.
AOO	44K	64K	69.9%	25.4%	1.5%	3.1%	0.6K	1.3K	195	0	0.05%
Linux	656K	787K	90.5%	6.6%	1.4%	1.5%	1.9K	3.3K	735	76	0.07%
Mozilla	51K	70K	83.1%	10.9%	2.0%	4.0%	2.0K	3.6K	694	39	0.26%
Quake	3K	5K	72.5%	23.5%	1.9%	2.0%	9	59	0	0	0%
VideoLAN	13K	15K	92.7%	5.5%	0.7%	1.1%	0.5K	0.6K	31	67	0.26%

Table III

DATA ABOUT USAGE OF PRESENCE CONDITIONS AND USAGE OF ERROR AND WARNING DIRECTIVES. DEFINITIONS/SYM = NUMBER OF SYMBOLS DEFINED AT LEAST ONCE, DEF = NUMBER OF `define`, D1 = RATIO OF SYMBOLS DEFINED ONCE, D4+ = RATIO OF SYMBOLS DEFINED FOUR OR MORE TIMES, UNDEFINITIONS/SYM = NUMBER OF SYMBOLS UNDEFINED AT LEAST ONCE, UNDEF = NUMBER OF `undef`. PERC. = PERCENTAGE OF ERROR AND WARNING DIRECTIVES AMONG ALL THE PREPROCESSOR STATEMENTS.

In general preprocessing symbols can be re-defined or undefined multiple times. For example in the Quake project the symbol `INTERFACE` is defined or undefined 46 times, `LOG` is defined or undefined 223 times in Mozilla and `pr_fmt` is defined or undefined 995 times in Linux. This happens for a variety of reasons. In some cases different definitions of the same symbols are contained in header files which are alternatively included in the compiled system. In other cases different definitions of the symbols are used in separated subsystems, so they just happen to have the same name but they are intended as different values to be used in different contexts.

In table IV we report the frequencies of the particular cases described in which we can equiparate symbols to constants. As we can see in all the projects considered the percentage of symbols that are covered by these special cases ranges between 95% and 99%. It means it is feasible to automatic lift a large portion of the symbols, while a minority of them have to be manually converted.

3) Addressing RQ3: Error and warning directives: We simply counted the number of `error` and `warning` directives used.

Results: Data is available in Table III. Only one project (Quake) do not use them at all. In general we can notice that `error` directives are more used than `warning` directives; this is true for 3 out of 4 projects, while in the VideoLAN project `warning` directive are used twice as much as `error` directives. In general these directives constituted between 0.05% and 0.26% of all the preprocessor statements, if they are used at all.

4) Discussion: From our results we can tell that:

- **RQ1** Presence conditions can contain a range set of different expressions. However mathematical and bitwise operations are rarely used, as well as macro function calls, so they have not to be necessarily supported in mbeddr: the expressions not supported (if found in the projects the user want to import) can be manually addressed.
- **RQ2** Most of preprocessing symbols are used in practice as global constants, being never redefined to a different value under the same presence condition and being never undefined, therefore our simplifications appear to be reasonable and would permit to lift in mbeddr most of the symbols.
- **RQ3** `error` and `warning` directives are not necessarily used by all projects. Therefore in some cases constraints between features have to be extrapolated from other information sources or be manually described.

These results suggest it is feasible to extract and lift automatically a large portion of the variability information from C projects, while limited human intervention can be still needed.

IV. EXPERIENCE WITH IMPORTING CHIBIOS INTO MBEDDR

ChibiOS is a real-time operating systems supporting 14 core architectures, different compilers and different platforms. We chose it for our case study because it is a well written, complex embedded system with very high usage of variability to support portability. In our case study we used the code of version 2.5.1. The system is composed from many modules:

Expr. Type	AOO	Linux	Mozilla	Quake	VideoLAN	Range
single definition	69%	90%	80%	72%	90%	69%-90%
re-definition to the same value	23%	6%	7%	24%	2%	2%-24%
definitions under different conditions	2%	2%	9%	2%	4%	2%-9%
total	94.8%	97.9%	96.3%	98.6%	95.1%	94.8%-98.6%

Table IV
SPECIAL CASES IN WHICH MACROS CAN BE LIFTED TO HIGHER LEVEL CONCEPTS.

- *boards*: it contains specific code for 35 boards and a board simulator,
- *demos*: it contains 42 demos for different boards and compilers combinations (some of the demos are then divided in sub-demos),
- *os*: the directory containing the core of the system,
- *test* and *testhal*: contain source code for testing the system under different configurations.
- *tools*: tools which complement ChibiOS.

Given this organization we segmented the global system in sub-systems. In this section we first discuss how we extracted a feature model from the OS Kernel module (IV-A), then how we extracted a configuration for that feature model from the subsystem containing the code of a demo for a particular platform (IV-B). Finally we discuss the experience (IV-C).

A. The OS Kernel module

We start our analysis from the *os/kernel* subsystem because it contains code that have to work with all the architectures, boards and compilers supported by the system. Thereby, this is the module where portability is more important. Examining the code we noticed that most of the presence conditions have the shape `'a-sub-expression' || defined(__DOXYGEN__)`. This has the goal of making visible pieces of code to a tool used to produce documentation. Because this symbol is not related to variability we substituted it with the value `'0'` (which evaluates to false for the preprocessor) and simplified the expressions containing it (for example `defined(A) || defined(__DOXYGEN__)` would become `defined(A)`). This permits to identify as redefinition guards snippets as the one presented in Listing 5.

```
#if defined(A) || defined(__DOXYGEN__)
#define A 123
#endif
```

Listing 5. A redefinition guard polluted by the __DOXYGEN__ symbol

We parsed correctly all the 41 files (18 C files and 23 header files). Excluding the statements described in Par. III-B2 we obtained 246 presence conditions expressions (all parsed correctly) and 233 definitions (both of symbols with object-like or function-like symbols). We examined which symbols were used in presence conditions: they were 54, none of them being a function-like symbol. We then examined all the definitions of these symbols in the subsystem to iteratively look for symbols that were indirectly referred by presence conditions. We found only the symbols `TRUE` and `FALSE` to be indirectly referred by presence conditions. Out of the total 56 symbols used (directly or indirectly) only 3 symbols were defined internally at the module: `CH_DBG_ENABLED`, `TRUE` and `FALSE`. Being the other 53 symbols never defined in the subsystem we know they have to be defined in the modules "using" the kernel module. For this reason we lifted them as features. We therefore created a feature model containing these 53 features.

The three symbols used in presence conditions and defined in the subsystem were instead imported as derived features. For `TRUE` and `FALSE` there was just one definition which was always valid (i.e., the definitions were not inserted in a variation point). `CH_DBG_ENABLED` instead had two definitions. The first one under the condition `CH_DBG_ENABLE_ASSERTS || CH_DBG_ENABLE_CHECKS || CH_DBG_ENABLE_STACK_CHECK || CH_DBG_SYSTEM_STATE_CHECK`, the second one under the opposite condition. In the first case it was defined to `TRUE`, in the second one to `FALSE`. We could import it automatically because the different conditions under which it was defined were disjoint, otherwise we would have needed human judgement to import it.

To complete the feature model we imported the extra constraints from warning and error statements. In the subsystem analyzed there were 6 errors statements and 0 warning statements. For each error we calculated the *presence conditions* and extracted the message.

B. Module demos/ARMCM3-STM32F103ZG-FATFS

The code of this module contains a definition for 31 out of the 53 features present in the feature model extracted by the OS Kernel. Other features could remain undefined or be defined in other modules which are compiled together with this one to produce the final demo: for example the module containing the board-specific code or the code specific for the ARM architecture. All the definitions of these features have a *presence condition* equals to true. It means they are always valid. The features have assigned in 28 cases either the value `TRUE` or `FALSE`. Note that those macros assume the value 1 or 0 (the preprocessor or the C language have not a boolean type). We decided to import them as booleans (which is supported by mbeddr-C) instead of simple integers. One of the remaining three features is defined without providing a value, while the other two assume the numerical values 0 and 20.

C. Discussion

ChibiOS is a system written extremely well and the usage of the preprocessor is very disciplinated. Because of that we were able to extract without human intervention a feature model and a configuration model from two of the modules of the whole system. We could extract some of the constraints of the feature model from `error` directives but most of them had to be manually specified. The type of values that can be associated to features was obtained indirectly, looking at how the symbol was used in the configuration and updating the feature model.

V. Related work

We classify the related work along four directions:

Substitute preprocessor directives Kumar et al. [7] discuss how to substitute some usages of preprocessor directives with features of the new standard of C++ (C++11). ASTEC [8] is a variant of C with support to syntactic macros. McCloskey et al. explain how they analyzed C code and refactor the preprocessor directives using these extensions. Both approaches target languages with no explicit support for variability. ASTEC in particular lift the level of abstraction from token to syntactic level.

Representing variability There are development tools which provide a more intuitive representation of variability concerns through the usage of background colors (e.g., [9]). Our work could be integrated with such tools.

Variability information extraction Some approaches aims to provide tools able to analyze the preprocessor directives and individuate possible bugs. Among these approaches one of the most relevant is TypeChef [6]. While the goal of these approaches is to achieve maximum accuracy and generality, our approach is to capture the intentions as expressed in the code.

Empirical study about preprocessor usage On this topic it is very relevant the analysis presented by Ernst et al. [10]. While their analysis is very deep and interesting, they considered less lines of code that we did and they did not address large projects as we did. The goal of their analysis is more general, while we focus specifically on variability. However we can confirm one of their findings: they reported that 86% of preprocessor symbols were defined just once among the 26 systems they considered. We found this value to range between 69% and 90%.

VI. Conclusions and future work

In this paper we examined how in practice large, long established projects use preprocessor directives to represent variability. Results suggest that this category of projects are quite disciplined in using them, as consequence idioms and patterns can be identified and exploited to extract a large part of the variability information present in the code using simple heuristics. Our approach aims to preserve the readability and the original intent expressed in the code: while it has some theoretical limitations, it seems to be applicable in practice, as suggested also by the results of our experience with ChibiOS.

Our approach have still to be improved and completed: we have to use the variability information extracted to decorate the statements with presence conditions. Possible improvements could include attributing automatically a type to the extracted features. As future work we plan to perform a case study on a project involving industrial partners.

Acknowledgments

We would like to thank all the contributors to the mbeddr project at Itemis and Fortiss for their support, Markus Völter for his suggestions and Christian Kästner for the useful discussions we had with him. We would like also to thank DAAD that partially financed this work. Mbeddr has been supported by the German BMBF, FKZ 01/S11014.

References

[1] K. Nie and L. Zhang, "On the relationship between preprocessor-based software variability and software defects," in *High-Assurance Systems Engineering (HASE), 13th Int. Symp. on*, 2011, pp. 178 –179.

[2] H. Spencer and G. Collyer, "#ifdef considered harmful, or portability experience with c news," in *Proc. of the Summer 1992 USENIX Conf.*, San Antionio, Texas, 1992, pp. 185–198.

[3] M. Voelter, D. Ratiu, B. Schaetz, and B. Kolb, "mbeddr: an extensible c-based programming language and ide for embedded systems," in *Proc. of the 3rd conf. on Systems, programming, and applications: software for humanity*, ser. SPLASH '12. New York, NY, USA: ACM, 2012, pp. 121–140.

[4] M. Voelter and E. Visser, "Product line engineering using domain-specific languages," in *Software Product Line Conference (SPLC), 15th Int.*, 2011, pp. 70 –79.

[5] D. Ratiu, M. Voelter, B. Schaetz, and B. Kolb, "Language Engineering as Enabler for Incrementally Defined Formal Analyses," in *FORMSERA'12*, 2012.

[6] A. Kenner, C. Kästner, S. Haase, and T. Leich, "Typechef: toward type checking #ifdef variability in c," in *Proc. of the 2nd Int. Workshop on Feature-Oriented Software Development*, ser. FOSD '10. New York, NY, USA: ACM, 2010, pp. 25–32.

[7] A. Kumar, A. Sutton, and B. Stroustrup, "Rejuvenating c++ programs through demacrofication," in *Proc. of the 28th IEEE International Conference on Software Maintenance*, 2012.

[8] B. McCloskey and E. Brewer, "Astec: a new approach to refactoring c," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 21–30, 2005.

[9] J. Siegmund, N. Siegmund, J. Fruth, S. Kuhlmann, J. Dittmann, and G. Saake, "Program comprehension in preprocessor-based software," in *Computer Safety, Reliability, and Security*, ser. Lect. Notes in Computer Science. Springer Berlin Heidelberg, 2012, vol. 7613, pp. 517–528.

[10] M. Ernst, G. Badros, and D. Notkin, "An empirical analysis of c preprocessor use," *Software Engineering, IEEE Transactions on*, vol. 28, no. 12, pp. 1146 – 1170, 2002.