Partial Preprocessing C Code for Variability Analysis

Christian Kästner, Paolo G. Giarrusso, and Klaus Ostermann
Philipps University Marburg
Marburg, Germany

ABSTRACT

The C preprocessor is commonly used to implement variability. Given a feature selection, code fragments can be excluded from compilation with #ifdef and similar directives. However, the token-based nature of the C preprocessor makes variability implementation difficult and error-prone. Additionally, variability mechanisms are intertwined with macro definitions, macro expansion, and file inclusion. To determine whether a code fragment is compiled, the entire file must be preprocessed. We present a partial preprocessor that preprocesses file inclusion and macro expansion, but retains variability information for further analysis. We describe the mechanisms of the partial preprocessor, provide a full implementation, and present some initial experimental results. The partial preprocessor is part of a larger endeavor in the TypeChef project to check variability implementations (syntactic correctness, type correctness) in C projects such as the Linux kernel.

1. INTRODUCTION

To implement variability in software product lines, developers often use conditional-compilation mechanisms of the C preprocessor *cpp* or similar tools. With #ifdef and #endif directives they frame code fragments that are conditionally excluded during the compilation process. Depending on the feature selection, which is provided as command line parameters or configuration file, the preprocessor *generates* different *variants* of the program, with or without these code fragments. Conditional compilation with the C preprocessor is a very simple form of implementing compile-time variability, easy to learn and broadly used in practice [7, 12, 32, 39].

Unfortunately, the C preprocessor has several properties that render it error prone and difficult to analyze, which is also reflected by strong criticism in literature [11, 12, 25, 37]:

1. The C preprocessor is *token-based* and uses lexical macros [6]; as such, it is oblivious to the underlying language. That is, while the preprocessor can be used language-independently, it can also easily introduce syntax errors, such as parenthesis mismatch, in the host language. The preprocessor has no mechanism to detect potential problems in the underlying code. Conversely, any non-heuristic syntactic or semantic analysis of code before preprocessing is even regarded to be impossible by some [33].

2. The evaluation of conditional-compilation directives in the C preprocessor is deeply *intertwined* with the *file-inclusion* mechanism (#include) and *macro* facilities (#define and #undef). Using conditional compilation, alternative expansions of macros can be defined; and macro definitions can influence the evaluation of conditional-compilation directives.

3. Conditional compilation is not only used to implement compile-time variability for features in the product-line sense or low-level portability issues, but also for *include guards*. An include guard is a pattern that prevents multiple inclusion of the same file, which uses the same conditional-compilation mechanisms and is difficult to distinguish from variability by tools.

These properties make it difficult to analyze code that was not already preprocessed (called unpreprocessed code or *pre-cpp code*). Nevertheless, there are many interesting questions a developer might want to answer about variability, such as, "When is a code fragment included?", "What are possible expansions of a macro?", "Into what possible results can macros modify this line?", "Are all possible variants syntactically correct or well-typed?", "Are there code fragments that are never used in any variant?", and many more. Some of these questions, we can approximate by ignoring the underlying language, for example, we could ask "Under which feature selection is Line 8 included?", but for many others, such as "Under which feature selection is variable x initialized?", we need a more precise analysis and need to deal with macros and the host language. These questions are difficult to answer for both humans [12, 37] and automatic analysis tools [5, 15, 27].

It would be easier (for humans and tools) to answer such questions with a more disciplined form of implementing variability, such as syntax macros [30, 46], compile-time if statements as in D,[1], tool-driven feature mapping [8, 20], framework-based implementations [3], and feature-oriented programming [2]. We believe that in the long run, developers should switch to better variability-implementation mechanisms than lexical preprocessors. However, as long as they

[1] http://www.digitalmars.com/d/2.0/version.html

VaMoS'11 January 27-29, 2011 Namur, Belgium

have not been picked up broadly, and, in the presence of vast amounts of legacy code, our goal is to improve the situations for variability implementations with the C preprocessor.

We design and implement a partial preprocessor that separates variability implementation from the difficulties caused by file inclusion and macro expansion, and, thus, makes a first step to enable such analysis. In a form of partial evaluation or staging, inspired by prior work on *cpp* symbolic execution [15, 26, 27], the partial preprocessor processes a pre-cpp C file, evaluates all file inclusion, and expands all macros, but keeps all variability. That is, after partial preprocessing, only conditional-compilation directives (#if, #endif) remain in the C file, while all #include, #define, and #undef statements are resolved and all macros are expanded. Alternatively, the output of the partial preprocessor can be seen as a stream of tokens, in which each token has a condition that describes under which feature selections it is included in the compilation process.

Partial preprocessing does not answer all the analysis questions raised above, but it builds a foundation for further analysis steps that address the remaining questions. The partial preprocessor is the first step in our TypeChef project [22] in building an analysis framework, which can *parse* and *type check* all variants of arbitrary legacy pre-cpp code, without preprocessing the code for each feature combination in isolation.

In summary, we contribute (1) a comprehensive description of the problem of analyzing variability in pre-cpp code, (2) an approach to variability analysis, based on partially preprocessing code by evaluating file inclusion and macros but not conditional compilation, (3) an implementation on top of a C preprocessor, and (4) an evaluation demonstrating practicality but also current limitations of our solution for several realistic C files.

2. THE C PREPROCESSOR

The C preprocessor provides three main features when preprocessing a file [17]:

- File inclusion (#include): The preprocessor replaces the #include directive with the content of the target file and continues preprocessing. File inclusion is the only way to express iteration, but inclusion depth has a fixed limit, thus preprocessing is guaranteed to terminate.

- Macro definition and expansion (#define, #undef): Macros can be defined (#define) to expand tokens in the subsequent output stream. When, during preprocessing, a token is found that equals a macro's name, this token is replaced by the expansion of the macro (in addition to object-like macros, function-like macros can also have parameters and are replaced accordingly). The expanded macro can contain tokens that are target further for macro expansion (though not recursively). Defining the same macro for a second time redefines (i.e., replaces) the macro expansion; #undef resets a macro to undefined. Macros are dynamically scoped, that is, a macro is expanded using the definition currently in scope, not one in scope at macro-definition time.

- Conditional compilation (#ifdef, #ifndef, #if, #elif, #else, #endif): The preprocessor evaluates a condition, typically by checking whether certain macros are defined or undefined. Depending on the result it either outputs the code between #if and #endif directive or replaces that code by empty lines. Note, "#ifdef X" is merely syntactic sugar for "#if defined(X)" and "#ifndef X" for "#if !defined(X)"; for the remainder of this paper, we use only the basic form.[2]

main.c

```
1  #include "lib.h"
2  #if defined(WITH_GUI)
3  #include "gtk.h"
4  #endif
5
6  #define NAME foo
7
8  #if defined(BIT64)
9  #define T long
10 #endif
11 #if defined(BIT16)
12 #define T short
13 #endif
14
15 #if defined(NAME)
16 T NAME() {
17   return 3;
18 }
19 #if defined(T)
20 int main() { ... }
21 #endif
22 #endif
```

lib.h

```
23 #if !defined(_LIB_H)
24 #define _LIB_H
25 extern int open(...);
26 #include "lib.h"
27 #endif
```

Figure 1: Interaction of preprocessor facilities

Already on their own, these mechanisms are known to challenge code comprehension [12, 25, 34, 37]: In the presence of conditional compilation, it can be difficult to follow the control flow. Macros can have surprising effects for developers, especially when a macro, of which the developer was not aware, is defined in indirectly included header files.

However, we think that the main difficulty of reasoning about pre-cpp code comes from combining and interleaving the three preprocessor facilities. In code examples as in Figure 1, it can be quite difficult to understand which token is included under which condition or what token is expanded into which other tokens. First, the inclusion in Line 1 includes the header file, which includes further macro definitions and inner includes. Second, the inclusion in Line 3 is only executed if feature WITH_GUI is defined by the user (or in previously included header files). Third, the macro T is defined with alternative expansions in Lines 9 and 12. Fourth, "#if defined(NAME)" in Line 15 evaluates to true, independent of features provided by the user, since NAME is defined

[2] In addition to file inclusion, macro expansion, and conditional compilation, the C preprocessor also supports user-specified error reporting (#error and #warning), line directives (#line), and compiler-specific extensions (#pragma). Those preprocessor directives can be integrated straightforwardly into the partial preprocessor as well and are, in fact, mostly supported by our implementation, but we ignore their discussion in this paper to simplify the description of the partial-preprocessor mechanisms.

unconditionally in Line 6. Fifth, token T in Line 16 expands to long, short, or even does not expand at all, depending on the feature selection; token NAME is always expands to foo. Sixth, in Line 19, "#if defined(T)" evaluates to true if T, BIT64, or BIT16 were defined by the user (or some header files); it is nested inside another #if directive. Finally, Lines 23, 24 and 27 show the common include-guard pattern; even if the header is included multiple times, Line 26 is included only once because "#if defined(_LIB_H)" evaluates to false in the second inclusion. In Line 26, the header file includes itself; although such direct recursion is not a common pattern (usually include guards protect against indirect recursion and multiple inclusion from different files), we use it to demonstrate how include guards work later on.

For many analyses, we want to reason about variability (e.g., "Under which feature selections is function foo defined?"). In contrast, we are usually not interested in file inclusion and macro expansions. These are just technical and practical necessities when programming in the C language, typically used to work around C's lack of constants and modules. The need to analyze the definition of macros in all code (including in header files) makes it difficult to answer even a simple questions like "Under which feature selections is Line 20 included?"

3. A PARTIAL PREPROCESSOR

To separate variability mechanisms for conditional compilation from file inclusion and macro expansion, a partial preprocessor should evaluate file inclusion and macro expansion, but leave conditional compilation intact.

3.1 Desired Output

Let us start with the desired output of the partial preprocessor. The partial preprocessor should produce a token stream in which each token has a *presence condition*, which is a formula that describes under which feature selections the token is included for compilation [8]. The token stream should have the following characteristics:

1. In the output, we want to clearly know the condition for each token under which the token is included in the compilation process. The presence condition *should only depend on user-defined input* (the feature selection), not on #define and #undef directives in the source code. For example, the condition for Line 20 in Figure 1 should be $\mathsf{def(BIT64)} \vee \mathsf{def(BIT16)} \vee \mathsf{def(T)}$,[3] to reflect that the line is included (a) if T is defined externally by a user (potentially providing the parameter "-D T") or (b) if T is defined in Line 9 or 12. Similarly, the condition "#if defined(NAME)" in Line 15 can be replaced by true, because it is always true, independent of user input.

2. Macros should be expanded. That is, NAME in Line 16 should be expanded to foo. Tokens that have alternative expansions are expanded to all of them. Hence, T in Line 16 is expanded to both long and short (each with different presence conditions) – actually, even not expanding is an option when neither feature BIT64 nor feature BIT16 is defined.

[3]For brevity, we use def(X) as abbreviation of defined(X) in presence conditions.

with $\alpha = \neg\mathsf{def(\_LIB\_H)}$ and $\beta = \mathsf{def(BIT64)} \vee \mathsf{def(BIT16)} \vee \mathsf{def(T)}$:

$$\mathit{extern}_{\alpha}\ \ \mathit{int}_{\alpha}\ \ \mathit{open}_{\alpha}\ \ (_{\alpha}\ \ \ldots_{\alpha}\ \ )_{\alpha}\ \ ;_{\alpha}$$
$$[\ldots \text{ body of gtk.h } \ldots]_{\mathsf{def(WITH\_GUI)}}$$
$$\mathit{long}_{\mathsf{def(BIT64)} \wedge \neg \mathsf{def(BIT16)}}\ \ \mathit{short}_{\mathsf{def(BIT16)}}$$
$$\mathsf{T}_{\neg \mathsf{def(BIT64)} \wedge \neg \mathsf{def(BIT16)}}$$
$$\mathsf{foo}_{\mathsf{true}}\ \ (_{\mathsf{true}}\ \ )_{\mathsf{true}}\ \ \{_{\mathsf{true}}\ \ \mathit{return}_{\mathsf{true}}\ \ 3_{\mathsf{true}}$$
$$;_{\mathsf{true}}\ \ \}_{\mathsf{true}}\ \ \mathit{int}_{\beta}\ \ \mathit{main}_{\beta}\ \ (_{\beta}\ \ )_{\beta}\ \ \{_{\beta}\ \ \ldots_{\beta}\ \ \}_{\beta}$$

(a) token stream

```
28 #if !defined(_LIB_H)
29 extern int open(...);
30 #endif
31 #if defined(WITH_GUI)
32 ... body of gtk.h ...
33 #endif
34
35 #if defined(BIT64) && !defined(BIT16)
36 long
37 #endif
38 #if defined(BIT16)
39 short
40 #endif
41 #if !defined(BIT64) && !defined(BIT16)
42 T
43 #endif
44  foo() {
45   return 3;
46 }
47 #if defined(BIT64) || defined(BIT16) || defined(T)
48 int main() { ... }
49 #endif
```

(b) textual output

Figure 2: Desired output of the partial preprocessor

3. File inclusions should be resolved the usual way.

In Figure 2a, we illustrate the desired token stream for our initial example. We use a conditional token stream internally, because it is the most general representation and easy to process by tools. Since presence conditions are typically shared by multiple tokens, they do not cause serious memory overhead. From the representation as conditional token stream, we can directly derive other equivalent representations that are easier to read by developers, such as C code that contains #if and #endif statements, as shown in Figure 2b.[4]

The produced token stream with presence conditions can be used for further analysis. For example, we can directly recognize the condition under which a token is included in the compilation process, or we can easily recognize the alternative return types of method *foo*. As part of our *TypeChef* project, we are currently developing a parser that takes the token stream as input and produces a parse tree that represents all variability in a single abstract syntax tree (cf. Sec. 5).

To illustrate function-like macros, we close with a second example in Figure 3: In this example, macro FUNC is always expanded, because, although there are alternative expansions, there is an expansion for every possible feature selection.

[4]For error reporting and other purposes, we also store the origin of each token (file, line, and potentially responsible macro) and produce according #line directives in the serialized output.

```
1 #ifdef A
2 #define FUNC(x) ((x)*(x))
3 #else
4 #define FUNC(x) ((x)+(x))
5 #endif
6 int k = FUNC(3);
```

(a) input

```
1 int k =
2 #if (defined(A))
3 ((3)*(3))
4 #endif
5 #if (!(defined(A)))
6 ((3)+(3))
7 #endif
8 ;
```

(b) partially preprocessed

Figure 3: Partial preprocessing of alternative function-like macros

3.2 Design

To preprocess a file only partially, we have to change the way a preprocessor processes the file. In a nutshell, the following main changes are necessary: (1) we derive a presence condition for every token we read, (2) we store alternative expansions of a macro and their presence condition, (3) we change the evaluation of #if directives to reasoning about satisfiability, and (4) we expand macros multiple times if necessary.

3.2.1 Presence Conditions

First, we need to constantly store the presence condition for each token we read. For the first token, the presence condition is $\mathtt{pc} = \mathsf{true}$ (it is parsed independent of any feature selection). When we encounter a directive "#if X" the presence condition for all following tokens is $\mathtt{pc}_{new} = \mathtt{pc} \wedge \mathsf{X}$ until the corresponding #endif directive.[5] For #if X_1-#elif X_2-#elif X_3-...-#elif X_n-#else-#endif chains, presence conditions are calculated as follows $\mathtt{pc}_i = \mathtt{pc} \wedge \mathsf{X}_i \wedge \bigwedge_{1 \leqslant j < i} \neg \mathsf{X}_j$ and $\mathtt{pc}_{else} = \mathtt{pc} \wedge \bigwedge_{1 \leqslant j < n} \neg \mathsf{X}_j$.

In addition to Boolean flags (defined or undefined macros), the C preprocessor also supports integer constants and arithmetic operations on them, such as "#if VERSION>(3+NEW)". Hence, presence conditions require a larger set of operations then propositional logic. For automated reasoning (see below), we still use propositional logic and SAT solvers, by evaluating subexpressions with integer constants as far as possible (which works in almost all cases in our experience, because, typically, all values of integer constants are defined in previous macro definitions). In the remaining cases, we can introduce new Boolean variables to reflect distinct ranges of possible integer values.

3.2.2 Macro Table

When we encounter the definition of a macro, we store the macro and the current presence condition in a macro table, as illustrated in Figure 4. In subsequent steps, the presence condition tells us under which condition the macro is defined and should be expanded.

[5]Actually, as we will show in Section 3.2.3, X in "#if X" is first normalized to refer only to user-defined features, so also the resulting presence condition refers to user-defined features only.

after #define _LIB_H with $\mathtt{pc} = \neg\mathsf{def}(\mathsf{\_LIB\_H})$:

Name	Expan.	Presence Condition
_LIB_H	∅	¬def(_LIB_H)

after #define NAME foo with $\mathtt{pc} = \mathsf{true}$:

Name	Expan.	Presence Condition
_LIB_H	∅	¬def(_LIB_H)
NAME	foo	true

after #define T long with $\mathtt{pc} = \mathsf{def(BIT64)}$:

Name	Expan.	Presence Condition
_LIB_H	∅	¬def(_LIB_H)
NAME	foo	true
T	long	def(BIT64)

after #define T short with $\mathtt{pc} = \mathsf{def(BIT16)}$:

Name	Expan.	Presence Condition
_LIB_H	∅	¬def(_LIB_H)
NAME	foo	true
T	long	def(BIT64) ∧ ¬def(BIT16)
T	short	def(BIT16)

after #undef T with $\mathtt{pc} = \alpha$ (not in Fig. 1):

Name	Expan.	Presence Condition
_LIB_H	∅	¬def(_LIB_H)
NAME	foo	true
T	long	def(BIT64) ∧ ¬def(BIT16) ∧ ¬α
T	short	def(BIT16) ∧ ¬α
T	*undef*	α

Figure 4: Macro Tables during Partial Preprocessing

The interesting part of this macro table is that macros may be redefined and undefined during preprocessing, potentially with different presence conditions. Where a traditional preprocessor would just replace or remove the macro definition, the partial preprocessor must be able to handle alternative expansions of a macro. Redefinitions are more likely in a partial preprocessor, because a partial preprocessor typically evaluates both branches of an #if-#else-#endif directive. For example, Line 12 in Figure 1 redefines macro T with a different presence condition.

Redefinitions are stored in the macro table as follows: The new expansion is added to the macro table with its presence condition $\mathtt{pc}_{newmacro}$, without overwriting the previous expansions. Additionally, the presence condition of all existing expansions is changed as follows $\mathtt{pc}_{new} = \mathtt{pc} \wedge \neg\mathtt{pc}_{newmacro}$. Afterward, the following two cleanups are possible: Any expansion with a presence condition that is a contradiction (determined with a SAT solver) can be removed from the macro table, because it can never be used to expand a macro. If a macro has two equivalent expansions, they can be joined with a common presence condition $\mathtt{pc}_1 \vee \mathtt{pc}_2$.

Undefinitions with #undef are handled just as redefinitions, with the only distinction that we add an *undefined marker* instead of a new expansion. When subsequently

evaluating #if directives, the undefined marker is necessary to distinguish macros that were explicitly undefined from macros that may or may not be defined by the user.

These rules naturally result in a table which contains all possible macro expansions and contains an exact specification of which expansion is to be used under which condition. It still recreates the behavior of normal preprocessors when macros are redefined or undefined with the same presence condition; for example, after redefinition the previous expansion is removed because its presence condition is changed to the contradiction $\mathrm{pc} \wedge \neg \mathrm{pc}$.

3.2.3 Evaluating #if Directives

The evaluation of #if directives changes significantly compared to traditional preprocessors. In traditional preprocessors the expression of the #if directive can be evaluated directly, because, for each macro, it is known whether this macro is defined (by a previous #define directive or by a command-line parameter provided by the user); so, the expression always evaluates to either true or false. In contrast, the partial preprocessor intends to keep variability. Nevertheless, some form of evaluation is necessary to detect code fragments that can never occur in any variant. Without such evaluation, the header in Figure 1 would recursively include itself and partial preprocessing would never terminate.

So how do we evaluate an expression, such as "defined(BIT16)" in Figure 1? First, we *normalize* the condition such that it refers to user-specified features only. For a subexpression defined(X) that checks the definition of macro X, we look up X in the macro table. If there are one or more expansions with the presence conditions $\mathrm{pc}_1, \mathrm{pc}_2, ..., \mathrm{pc}_n$, we replace defined(X) with $\mathsf{def(X)} \vee \bigvee_i \mathrm{pc}_i$, a disjunction of these presence conditions (note that, unless $\bigvee_i \mathrm{pc}_i$ is a tautology, def(X) is still necessary to allow the possibility that X was defined by the user). If there is an undefined marker with presence condition $\mathrm{pc}_{\mathtt{undef}}$ in the macro table to indicate that the macro was explicitly undefined in some cases, the subexpression is replaced by $(\mathsf{def(X)} \vee \bigvee_i \mathrm{pc}_i) \wedge \neg \mathrm{pc}_{\mathtt{undef}}$.[6]

After normalization, we check whether the resulting expression is *satisfiable*, that is, whether there is at least one user-specified feature selection for which the source code will be included. If the expression is satisfiable, we continue processing with the new presence condition. If it is not satisfiable (i.e., a contradiction), we do not further preprocess any input up to the closing #endif directive. Hence, instead of checking whether the expression evaluates to true or false given the current macro definitions, we check whether the expression could be true at all with any feature selection, under the restrictions provided by the current macro definitions. Any code that could be generated for any possible feature selection will be generated by the partial preprocessor.

This kind of evaluation implies some interesting consequences; it handles #define and #undef directives elegantly and subsumes include guards without additional overhead or heuristics. A top level #define X directive inserts an expansion with presence condition true into the macro table. Following #if X directives are satisfiable (actually even tautologies). A top level #undef X directive inserts a undefined marker with presence condition true so that subsequent #if X directives are always contradictions. Include guards are handled naturally, which we show again on the example of Figure 1. First, the partial preprocessor checks "#if !defined(_LIB_H)" in Line 23. Since macro _LIB_H is neither defined nor explicitly undefined, we cannot replace it with a presence condition from the macro table; hence, the expression is satisfiable and preprocessing continues. Next, macro _LIB_H is defined with the current presence condition ¬def(_LIB_H) in Line 24. When the header is recursively included, we need to evaluate "#if !defined(_LIB_H)" again. This time, we find an expansion in the macro table and replace the condition as follows $\neg\big(\mathsf{def(\_LIB\_H)} \vee \neg\mathsf{def(\_LIB\_H)}\big)$, which is a contradiction. So, we stop recursive preprocessing of the header file; the include guard has prevented multiple inclusion.

[6]Note, we do not need to look up macro definitions for this result again, because presence conditions in the macro table are already normalized and refer to user-defined features only.

```
50 #if defined(VERSION1)
51 #define VERSION 1
52 #elif defined(VERSION2)
53 #define VERSION 2
54 #else
55 #define VERSION 3
56 #endif
```

Figure 5: Specifying possible expansions of numeric macros

3.2.4 Expanding Macros

The partial preprocessor expands macros like traditional preprocessors, but, again, it is able to handle alternative expansions. A token that matches a macro definition is replaced by all expansions, in which each expansion is wrapped in an #if directive with the corresponding presence condition. For example, in Figure 2, T is expanded to long and short, each with the corresponding presence condition.[7] If the disjunction of the presence conditions of all expansions is not a tautology (i.e., if there is a possible user-specified feature selection in which the macro is not defined), the unexpanded macro remains as alternative to the expansions with a corresponding presence condition.

Note that macro expansion can also occur inside the condition of #if directives (e.g., "#if VERSION>3"). For macros with alternative expansions, we introduce a specialized *if* construct (e.g., "#if IF(defined(NEW),4,3)>3") which is later flattened (e.g., "#if (defined(NEW) && (4>3)) || (!defined(NEW) && (3>3))") and simplified. Note that the C preprocessor implicitly assumes undefined tokens in #if expressions as 0; the partial preprocessor issues a warning when this occurs. We recommend a pattern that defines all valid expansions, as exemplified in Figure 5, as a way of reducing numeric parameters to Boolean variables.

3.3 Partial Configurations and Variability Models

The partial preprocessor is directly capable of handling *partial configurations* and constraints of a *variability model*. In a configuration file, we can simple define and undefine all features of a partial configuration (with normal #define

[7]Although it is theoretically possible to mix object-like style and function-like style of macro expansion in alternatives, this only occurred rarely in one of our case study. For simplicity, our implementation currently expects all alternative expansions of a macro to have the same number of arguments.

and #undef statements). The partial preprocessor adds these features to the macro table and considers them defined or undefined with presence condition true (i.e., not user-specified) for all further preprocessing steps. For example, in Figure 1, we could provide a partial configuration in which *BIT64* is defined; so, the line "#define T long" would be reached with the presence condition true and "#define T short" would not even be reached during preprocessing; subsequently, T would always expand to *long*.

We can use partial configurations to reduce complexity during partial preprocessing. For example, we can initially undefine all macros used for include guards (e.g., by pattern matching "_*_H"), so that they are no longer part of presence conditions; it appears reasonable to assume that a user would not select include-guard macros as features.

To handle variability models, we can either encode dependencies between features with #if, #define, and #undef directives (for example "#if defined(A) #define B #endif" to denote the constraint "feature A implies feature B"), or we can pass them as propositional formula VM that is used in all satisfiability checks (VM ⇒ Condition, i.e., is the condition satisfiable under the valid configurations specified by the feature model). Most variability-modeling notations can be transformed into propositional formulas for such encoding [4].

3.4 Implementation

We have implemented the partial preprocessor with Java and Scala on top of *jcpp*,[8] an existing Java implementation of the C preprocessor . For reasoning about #if conditions, we have implemented a library of feature conditions that we can check for satisfiability (and tautologies and contradictions) using the off-the-shelf SAT solver sat4j.[9] The implementation is publically available in the repository of the TypeChef project.[10]

The implementation follows the general strategy outlined above: When preprocessing a file, the partial preprocessor tracks the current presence condition. To evaluate a conditional-compilation directive, we run the SAT solver to determine whether the condition is satisfiable and produce output accordingly. For macros, we store alternative expansions and revise their presence conditions each time when #define and #undef directives are encountered. When expanding a token that has multiple expansions (with satisfiable conditions), we replace the token by all expansions, wherein each expansion is wrapped in its own conditional-compilation directive.

To test correctness, we simply run the standard C preprocessor on the output of the partial preprocessor. For every feature configuration, the preprocessor should yield the same token stream when executed (a) on the original C file and (b) on the partially preprocessed file.

4. EVALUATION

To partially preprocess a file requires more computations than preprocessing that file for a specific feature selection, because all possible variants are produced. During partial preprocessing, our preprocessor frequently determines possibly complex presence conditions and their satisfiability (which itself is NP-hard in principle). By partial preprocessing, even simple code fragments can explode to long output sequences, for example, when a token has many alternative expansions. Still, in several experiments, we found that partial preprocessing is tractable for many realistic C files. In addition, several researchers confirmed that SAT solving is fast for most feature-related problems in product-line development [31, 44]. Hence, we argue that, instead of theoretical assessment of worst case performance, we should evaluate practical (average case) performance empirically.

To demonstrate practicability, we assess the following questions on several example files and systems:

- What is the shape of common preprocessor usage (e.g., how common are alternative macro definitions)?
- What performance can be expected from a partial preprocessor?
- How large is a partially preprocessed file (i.e., is it realistic to precompute all variability)?

The partial preprocessor is part of ongoing research effort (cf. Sec. 5); hence, we have applied it only to some selected case studies so far. We select the following case studies:

- The small web server *Boa* (6 200 LOC, 120 features), which we already analyzed with a prior version of a partial preprocessor [22].
- The bug finding tool *sparse* (33 400 LOC, 4 features), which is relatively fresh C code (developed since 2003) with very few #if directives; variability is mainly caused by included header files of the system or the compiler.
- The text editor *Vim* (385 800 LOC, 778 features), as a representative of a project that contains unusually many and complex #if directives, as found during a previous analysis [28].
- Selected files from the Linux kernel (total of about 12 million LOC and 8000 features [36]); we sampled files in an ad-hoc fashion because the build system is complex and we have not automated the application of the partial preprocessor in that context, yet. For some files, partial preprocessing did not terminate (e.g., init/main.c, arch/x86/kernel/signal.c, kernel/fork.c); see discussion below. For others, partial preprocessing posed no problem. Our statistics are based on ten files that our partial preprocessing could process sucessfully (init/calibrate.c, arch/x86/kernel/irq.c, arch/x86/kernel/setup.c, arch/x86/kernel/ioport.c, kernel/kprobes.c, kernel/exit.c, lib/proportions.c, lib/prio_tree.c, mm/filemap.c, and mm/oom_kill.c).

For our experiments, we used headers from the *gcc* compiler version 4.4.4 and *Fedora* 13.

4.1 Preprocessor Usage

In all projects, a preprocessor has to include a large number of header files with a large number of macros (or features). Each file includes 31 to 710 distinct header files, each with 500 to 13 000 macro definitions. Of these macro definitions, only few (up to 70, and up to 700 in *Linux*) have alternative expansions, but quite a substantial number are defined with a presence condition other than true (16 to 10 000). In Table 1, we show individual results (minimum value, median, and maximum value) for each analyzed project.

[8] http://www.anarres.org/projects/jcpp/
[9] http://www.sat4j.org/
[10] http://github.com/ckaestne/TypeChef/

These numbers show that, although alternative macro expansions are not very frequent,[11] they occur in all analyzed projects. Macros defined with presence conditions are quite common; for example, in *Vim* many macros belong to *gui* features (from *X* and *GTK* headers) that are included only in some variants. Preprocessing involves many deeply nested header files.

As a consequence of this complexity, we argue that analysis and error detection are important and that the partial preprocessor provides a first step to enable such analysis.

4.2 Performance

Running the partial preprocessor naively on a C file often will *not* terminate within reasonable time (we did not wait more than 30 minutes per file; in problematic cases, we usually run out of memory before). The problem lies in typical header files provided by the environment (by the operating system or compiler) that recursively include other header files, each with their own include guards. A nesting depth of 10 to 20, each file with an include guard, and long #if-#elif chains are not uncommon, but lead to large expressions for presence conditions and in the macro table. Frequently determining satisifiability of such complex expression slows down partial preprocessing to the point that it becomes unusable (just the transformation of the formula into an equisatisfiable conjunctive normal form required by the SAT solver can take minutes for large expressions).

However, by providing a partial configuration in which all macros that are used only in include guards are undefined (cf. Sec. 3.3), presence conditions become smaller and most SAT problems become tractable. Partially preprocessing a file from *Boa* takes 1.8 seconds on average (on a 3 GHz dual core machine with 2 GB of ram), a file in *sparse* requires between 2 and 12 seconds, several *Linux* files require 4 to 8 seconds.

These results are encouraging and demonstrate that that the partial preprocessor scales for many realistic cases.

Nevertheless, with *Vim* and some files from *Linux*, we reach the limits of our current implementation. In *Vim*, there are many and partially complex conditions and an entire feature model (with many dependencies between features) is encoded using #if and #define directives. Without the feature model, preprocessing requires between 20 and 70 seconds for most files and did run out of memory for 2 out of 56 files. However, when considering the feature model naively without optimization, only few files terminate within reasonable time at all. Also with several Linux files that conditionally include many headers (at least the three files init/main.c, arch/x86/kernel/signal.c, and kernel/fork.c, mentioned above), we currently have performance problems. The partial preprocessor spends most of the time on determining satisfiability, of which the expensive part is to derive a formula in conjunctive normal form that can be fed into the SAT solver (the time actually spent by the SAT solver is negligible). Nevertheless, we are confident that we can solve the performance bottleneck with optimized implementations using strategies similar to those we used for reasoning about feature models with up to 10 000 features [44].

Another possibility to improve performance is caching. Although we have not implemented caching yet because simply undefining macros of include guards with a pattern expression has served us well so far, the partial preprocessor actually allows very precise caching of include directives. For a given macro table, an #include directive will always yield the same output and the same resulting macro table. (We can even determine a partial macro table with all macros relevant for that header file, so we could independently cache header files that do not influence each other.) Hence, we can cache the result of file inclusion for a given (partial) macro table, even when preprocessing very different input files, whereas contemporary C compilers cache at most the first header file. In several projects, especially in *Vim*, caching appears promising, since all files essentially include the same headers. Determining the effectiveness (hit ratio, time saving) of such cache is an interesting question for future research.

In the worst case, we could build the partial preprocessor without reasoning about satisfiability at all, if we find some (possibly project-specific) reliable mechanism to handle include guards. For example, we could exploit naming conventions and the their typical pattern in files. Drawbacks are that, without reasoning about satisfiability, we might generate dead code during macro expansion and the output might be incorrect in cases the include-guard heuristic fails. So far, we are confident and the preliminary results are encouraging, that the exact form of partial preprocessing as outlined in this paper is feasible in practice.

4.3 Size

The output of the partial preprocessor is typically 1.4 to 5 times the size (in lines of code, not counting empty lines) of running the preprocessor with a default configuration. For example, in Boa, the 259 LOC file *buffer.c* results in a partially preprocessed file with 11 001 LOC compared to 3 163 LOC after normal preprocessing. Such increase in size is to be expected. Preprocessed results are always much larger compared to the initial C code, because all headers are included. In addition, partial preprocessor produces all possible expansions instead of just a single one, so the result is larger again. Nevertheless, we consider also these results as encouraging, because the increase in output size is actually manageable; it does not explode into gigabytes of output that would be impractical to analyze further.

4.4 Perspective

In these case studies, different kinds of preprocessor usage become apparent. On one end of the spectrum is *sparse*, of which the entire implementation contains 34 #if directives. Sparse is not intended as software product line, but still, a large amount of complexity is introduced in sparse by variability in header files. On the other end of the spectrum is *Vim*, which provides a fine-grained compile-time variability with 12 652 #if directives. This amount of variability brings our current implementation of the partial preprocessor to its limits. We assume that most product line implementations will occur within this spectrum and can be partially preprocessed with acceptable performance.

5. RELATED WORK

5.1 Analysis of Preprocessor Directives

There have been many approaches to analyze the C preprocessor and its variability. Closest to our approach are

[11] The *Linux* kernel is a notable exception, in which code guidelines encourage to encode variability in alternative functions and alternative macro definitions, rather than conditional calls.

Case Study	C Files	Included Header Files (min, median, max)	Number of Macros (min, median, max)	Macros w/Alt. Exp. (min, median, max)	Conditionally Def. Macros (min, median, max)
Boa	21	66, 88, 99	1800, 2193, 2382	0, 8, 9	42, 48, 1621
sparse	45	31, 48, 526	543, 797, 5709	0, 0, 40	16, 35, 619
Vim	54	39, 643, 710	876, 10912, 13339	0, 56, 70	31, 7421, 9863
Linux	7	181, 333, 485	3936, 7015, 12685	250, 631, 701	831, 1197, 2249

Table 1: Case-Study Metrics

Hu et al. and Latendresse, who use symbolic execution to rewrite #if conditions [15, 26, 27]. Similar to our work, they analyze the interaction of #define and #if directives and replace #if conditions by conditions that only depend on to user-specified values. Hu et al. provide some example fragments from the Linux kernel but are unspecific about the performance and scalability of their approach [15]. Latendresse focuses on describing symbolic execution *formally* with a rewrite system [27], but evaluates applicability to practical problems only on a single C file and an individual header file (kernel.h) from the Linux kernel [26]. Our partial preprocessor differs from both previous works in that we also store and expand alternative macros during preprocessing. By using a SAT solver to determine whether the presence condition is still satisfiable, we provide a novel and elegant solution to recursive inclusion and include guards. Furthermore, we provide some initial evaluation to which degree our approach scales to realistic problems.

Sincero and Tartler et al. have taken a different approach to extract variability from pre-cpp code [41, 42]. They evaluate a C file (and optionally all included headers) and derive presence conditions for each *line* of code. However, they consider neither macro expansion nor interaction between #define and #if directives; that is, there is no distinction between user-defined features and macros defined within the source code. With their extracted variability they pursue the goal to derive a variability model of the implementation [41] and to detect dead code fragments [42].

Other forms of preprocessor analysis were proposed by Krone and Snelting, who used formal concept analysis to derive relationships between #define and #if directives; however, several aspects as macro expansion and alternative includes cannot be handled. With a different focus, Livadas and Small [29] and Spinellis [38] trace names in macro expansions, but do not handle #if directives and alternative macros.

5.2 Parsing Pre-CPP Code

There is a whole group of approaches, which go beyond analyzing only the preprocessor commands, but actually attempt to parse pre-cpp code. Instead of reasoning about lines or tokens and their presence conditions, as the approaches above, they reason about functions or statements in C and their variability. There are several different strategies to parse pre-cpp code for analysis [1, 5, 21, 33] and refactoring [13,14,30,45]. Several approaches use heuristics [13,14,33], which is shown to work reasonably well, but which might not be reliable enough for many cases, because it can lead to incorrect analysis results and incorrect code output. Other approaches limit the possibilities of how the preprocessor can be used [1, 5, 21, 30]. For example, instead of wrapping arbitrary tokens, #if and #endif directives may only wrap structural elements of the underlying language, such as entire functions or statements [5, 21, 30]. Vittek even uses a strategy to derive all possible preprocessor results (excluding variability in headers) [45], which does not scale when many #if expressions are used. Although the partial preprocessor only prepares files for parsing, it does so without heuristics and without restrictions on how the preprocessor can be used. We believe that it provides a solid foundation for parsing and analysis attempts.

Our partial preprocessor is the first step of the larger *TypeChef* (short for *type checking #ifdef variability*) project [22], which attempts parsing pre-cpp code. The goal is to check all variants of arbitrary C programs for syntax errors and type errors, such that all (potentially millions of) variants generated by the C preprocessor with different feature selections are well-typed. Obviously simply generating and compiling all variants does not scale (for n features there can be up to 2^n variants)Therefore, *TypeChef* aims to parse pre-cpp code into a single AST that keeps all variability, after which existing analysis approaches can be used, such as product-line–aware type systems [10, 16, 19, 43].

In *TypeChef's* parsing process, the partial preprocessor separates variability information from other facilities of the C preprocessor, such as file inclusion and macro expansion. In our outline of the *TypeChef* project [22], we previously provided a very simple partial preprocessor based on commenting out #if directives, running the original preprocessor and then removing the comments again. However that solution could not handle alternative macros and the interaction between #define and #if directives and lead to many false positives. Our solution in this paper can finally handle the full complexity of the C preprocessor. In ongoing work in the *TypeChef* project, we build a parser framework that builds a variability-aware abstract syntax tree out of the partial preprocessor's token-stream output.

5.3 Metaprogramming

The C preprocessor adds lightweight metaprogramming facilities to the C language [23]. Preprocessing can be seen as a (limited) form of multi-staged programming [40]. The preprocessor provides a lightweight metalanguage to perform (rather restricted) compile-time computations to generate C code. The goal of the partial preprocessor is to execute the metalanguage to some degree to specialize the program by evaluating and unfolding certain preprocessor directives, which is a form of partial evaluation [18]. Compile-time modifications would also be possible with many other metaprogramming facilities, such as template metaprogramming [9] or advanced macro systems [24, 46]. However, these approaches are bound to the syntactic structure of the host language and are typically more expressive and hence more difficult to analyze and partially evaluate than the lightweight mechanisms of

the C preprocessor. To the best of our knowledge there is no approach to evaluate a metaprogram only partially, in order to evaluate some but not all metalanguage constructs. Although some challenges, such as analyzing metaprograms and reporting errors at the right position in the user-written code, are similar in many metaprogramming facilities, the partial preprocessor is a much more tailored solution, which exploits the relative simplicity (despite all problems) of preprocessor directives. For example, instead of using heuristics or a general purpose theorem prover, we can use automated reason about preprocessing steps with a SAT solver.

Finally, one could imagine to translate #ifdef directives into if statements of the host language and macros into functions, which Post and Sinz outline as *lifting* [35], to subsequently apply standard analysis and transformation tools. In such setting, even specialization at runtime are imaginable. Unfortunately, for arbitrary input, such transformation from #ifdef to if is far from trivial and at least as difficult to automate as parsing pre-cpp code.

6. CONCLUSION

Variability analysis of product-line implementations is an interesting field to answer developer questions or to detect implementation bugs. However, when variability is implemented with the C preprocessor or similar tools, conditional compilation directives are intermixed with macro facilities and file inclusion, which makes analysis difficult. With a partial preprocessor, we provide a solution that evaluates macros and file inclusion but retains all variability for further analysis. The partial preprocessor also deals with complicated cases, such as alternative expansions of macros, and handles inclusion guards in an automated way without heuristics.

The partial preprocessor provides a first step toward variability analysis in legacy C code. As part of the *TypeChef* project, we are currently implementing a parser that can recognize a single abstract syntax tree that represents all variability from the partial preprocessor's token stream; on top of that abstract syntax tree, we are building a product-line–aware type system. Our long-term goal is to type check all variants of the Linux kernel without preprocessing every variant in isolation.

7. REFERENCES

[1] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. Can We Refactor Conditional Compilation into Aspects? In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 243–254. 2009.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[3] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[4] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 7–20. 2005.

[5] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. 2001.

[6] C. Brabrand and M. I. Schwartzbach. Growing Languages with Metamorphic Syntax Macros. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 31–40. 2002.

[7] P. Clements and C. W. Krueger. Point/Counterpoint: Being Proactive Pays Off/ Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, 2002.

[8] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 422–437. 2005.

[9] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[10] K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 211–220. 2006.

[11] M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.

[12] J.-M. Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop on Program Comprehension*, page 29. 1997.

[13] A. Garrido and R. Johnson. Refactoring C with Conditional Compilation. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, page 323. 2003.

[14] A. Garrido and R. Johnson. Analyzing Multiple Configurations of a C Program. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 379–388. 2005.

[15] Y. Hu, E. Merlo, M. Dagenais, and B. Laguë. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 196–206. 2000.

[16] S. S. Huang, D. Zook, and Y. Smaragdakis. cJ: Enhancing Java with Safe Type Conditions. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 185–198. 2007.

[17] International Organization for Standardization. *ISO/IEC 9899-1999: Programming Languages—C*, 1999.

[18] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, 1993.

[19] C. Kästner and S. Apel. Type-checking Software Product Lines – A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267. 2008.

[20] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. 2008.

[21] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, pages 175–194. 2009.

[22] A. Kenner, C. Kästner, S. Haase, and T. Leich. TypeChef: Toward Type Checking #ifdef Variability in C. In *Proceedings of the Second Workshop on Feature-Oriented Software Development (FOSD)*, pages

25–32. 2010.

[23] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, 1988.

[24] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic Macro Expansion. In *Proc. Conf. LISP and Functional Programming (LFP)*, pages 151–161. 1986.

[25] M. Krone and G. Snelting. On the Inference of Configuration Structures from Source Code. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 49–57. 1994.

[26] M. Latendresse. Fast Symbolic Evaluation of C/C++ Preprocessing using Conditional Values. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 170–179. 2003.

[27] M. Latendresse. Rewrite Systems for Symbolic Evaluation of C-like Preprocessing. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 165–173. 2004.

[28] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. 2010.

[29] P. E. Livadas and D. T. Small. Understanding Code Containing Preprocessor Constructs. In *Proc. Int'l Workshop on Program Comprehension (IWPC)*, pages 89–97. 2002.

[30] B. McCloskey and E. Brewer. ASTEC: A New Approach to Refactoring C. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 21–30. 2005.

[31] M. Mendonça, A. Wąsowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 231–240. 2009.

[32] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Proc. Int'l Conf. Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays)*, pages 313–329. 2002.

[33] Y. Padioleau. Parsing C/C++ Code without Pre-Processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125. 2009.

[34] T. T. Pearse and P. W. Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 270–277. 1997.

[35] H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. 2008.

[36] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. The Variability Model of The Linux Kernel. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 45–51. 2010.

[37] H. Spencer and G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, pages 185–198. 1992.

[38] D. Spinellis. Global Analysis and Transformations in Preprocessed Languages. *IEEE Trans. Softw. Eng. (TSE)*, pages 1019–1030, 2003.

[39] A. Sutton and J. I. Maletic. How We Manage Portability and Configuration with the C Preprocessor. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 275–284. 2007.

[40] W. Taha and T. Sheard. Multi-Stage Programming with Explicit Annotations. In *Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 203–217. 1997.

[41] R. Tartler, J. Sincero, D. Lohmann, and W. Schröder-Preikschat. Efficient Extraction and Analysis of Preprocessor-Based Variability. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 33–42. 2010.

[42] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD)*, pages 81–86. 2009.

[43] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 95–104. 2007.

[44] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 254–264. 2009.

[45] M. Vittek. Refactoring Browser with Preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110. 2003.

[46] D. Weise and R. Crew. Programmable Syntax Macros. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 156–165. 1993.