Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation

Christian Kästner   Paolo G. Giarrusso   Tillmann Rendel
Sebastian Erdweg   Klaus Ostermann

Philipps University Marburg, Germany

Thorsten Berger

University of Leipzig, Germany

Abstract

In many projects, lexical preprocessors are used to manage different variants of the project (using conditional compilation) and to define compile-time code transformations (using macros). Unfortunately, while being a simple way to implement variability, conditional compilation and lexical macros hinder automatic analysis, even though such analysis is urgently needed to combat variability-induced complexity. To analyze code with its variability, we need to parse it without preprocessing it. However, current parsing solutions use unsound heuristics, support only a subset of the language, or suffer from exponential explosion. As part of the TypeChef project, we contribute a novel variability-aware parser that can parse almost all unpreprocessed code without heuristics in practicable time. Beyond the obvious task of detecting syntax errors, our parser paves the road for further analysis, such as variability-aware type checking. We implement variability-aware parsers for Java and GNU C and demonstrate practicability by parsing the product line MobileMedia and the entire X86 architecture of the Linux kernel with 6065 variable features.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors; D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Algorithms, Languages, Performance

Keywords parsing, C, preprocessor, #ifdef, variability, conditional compilation, Linux, software product lines

1. Introduction

Compile-time variability is paramount for many software systems. Such systems must accommodate optional or even alternative requirements for different customers. In software product lines, variability is even regarded as a core strategic advantage and planned accordingly [51]. Unfortunately, variability increases complexity because now many variants of a system must be developed and maintained. Hence, many researchers pursue a strategy to lift automated analysis and processing—such as dead-code detection, type checking, model checking, refactoring, reengineering, and many more—from individual variants to entire software product lines in a variability-aware fashion [4, 13, 15, 24, 33, 52, 61].

In practice, a simple and broadly used mechanism to implement compile-time variability is *conditional compilation*—typically performed with lexical preprocessors such as the C preprocessor [30], Pascal's preprocessor [58], Antenna for Java ME [20], pure::variants [11, 53], and Gears [12]. Without loss of generality, we focus on the C preprocessor. A code fragment framed with #ifdef X and #endif directives is only processed during compilation if the flag X is selected, for instance because it is passed as configuration parameter to the compiler (as command-line option or using a configuration file). Using product-line terminology, we refer to such flags as *features* and to products created for a given *feature selection* as *variants*. Conditional compilation is widely used in open-source C projects [39], and it is a common mechanism to implement software product lines [51]; examples include Linux with thousands of features [39, 54], HP's product line of printer firmware with over 2000 features [49], or NASA's flight control software with 275 features [22].

One of the main problems of lexical preprocessors is that we cannot practically *parse* code without preprocessing it first. It is a common perception that parsing unpreprocessed code is difficult or even impossible [46, 58]. The inability to parse unpreprocessed code poses a huge obstacle for applying variability-aware analysis and processing to code bases using lexical preprocessors, such as the vast amount of existing C code. Parsing is difficult, because lexical preprocessors are oblivious to the underlying host language and its structure. Hence, conditional compilation can be applied to arbitrary token sequences, i.e., it is not restricted to syntactic structures. In addition to conditional compilation, lexical macros (again oblivious to the underlying structure), file inclusion, and

their interaction with conditional compilation complicate the picture.

Parsing, analyzing, and processing unpreprocessed code is interesting for many tasks involving variability, such as variability-aware error detection [4, 15, 33, 52, 57, 58, 61], program understanding [28, 36], reengineering [16, 54], refactorings [1, 23, 24, 64], and other code transformations [6, 46]. Variability-aware analysis and processing are especially important when the number of features grows, because there are up to 2^n variants for n features. Without suitable tools, developers typically analyze and process only few selected variants that are currently deployed. This way, even simple syntax or type errors may go undetected until a specific feature combination is selected, potentially late in the development process, when mistakes are expensive to fix.

Current attempts to parse unpreprocessed C code are unsound, incomplete, or suffer from exponential explosion even in simple cases. For instance, parsing all variants in isolation in a brute-force fashion does not scale for projects with more than a few features, due to the exponential number of variants. Alternatives use either unsound heuristics (such as assuming that macro names can be identified by capitalized letters) or restrict the way preprocessor directives can be used. Although such limitations are acceptable for some tasks or small projects, our goal is a sound and complete parsing mechanism that can be used on existing code to ensure consistency of an entire product line.

We contribute a novel variability-aware parser framework that can accurately parse unpreprocessed code. With this framework, we have implemented a sound and almost complete parser for unpreprocessed GNU-C code and a sound and complete parser for Java with conditional compilation. In contrast to existing approaches, parsers written with our framework usually *do not require manual code preparation* and *do not impose restrictions on possible preprocessor usage*. Our variability-aware parsers can handle conditional compilation even when it does not align with the underlying syntactic structure. In addition, we provide a strategy to deal with macros and file inclusion, using a variability-aware lexer. As such, our parsers can be used on existing legacy code. Without committing to a single variant, we detect syntax errors in all variants and create a parse result that contains all variability, so further analysis tools (such as type checkers or refactoring engines) can work on a common representation of the entire product line.

Although the worst case time and memory complexity of our parser and the produced abstract syntax tree are exponential, in practice, most source code is well-behaved and can be parsed efficiently. We use SAT solvers during lexing and parsing to efficiently reason about features and their relationships. In addition, we avoid accidental complexity by making decisions as local as possible. We demonstrate practicality with two case studies—a small Java product line *MobileMe-*

dia and the entire X86 architecture of the *Linux kernel* with 9.5 million lines of code and 6065 features.

The parser is part of our long-term project *TypeChef* (short for *type ch*ecking if*def* variability) but can also be used in isolation. We open sourced the entire implementation, provide a web version for easy experimentation, and publish additional data from our case studies at `http://fosd.net/TypeChef`.

In summary, we make the following novel contributions:

- We present a reusable variability-aware parser framework to parse code with conditional compilation, using SAT solvers for decisions during the parsing process. This framework is language-agnostic and can be reused for many languages using lexical preprocessors.
- We developed variability-aware parsers for Java and GNU C; to the best of our knowledge, these are the first parsers that can parse unpreprocessed code without excessive manual code preparation and without unsound heuristics.
- We demonstrate practicality of our parsers by parsing the product line MobileMedia and the entire X86 architecture of the Linux kernel.

Our contributions cover both (a) developing the novel concept of variability-aware parsing and (b) significant engineering efforts combining the parsers with other prior research results (i.e., reasoning about variability and variability models, variability-aware type systems, variability-aware lexing) to build a tool infrastructure that scales to parsing Linux.

2. Variability-aware parsing: What and why?

Our goal is to build a parser that produces a single abstract syntax tree for code that contains variable code fragments (optional or alternative). In the resulting abstract syntax tree, the code's variability is reflected in optional or alternative subtrees.

We illustrate the problem and the desired result on two simple expressions in Figure 1, in which the C preprocessor is used on numeric expressions. The preprocessor prepares the code by removing code fragments between #if and #endif directives if the corresponding feature is not selected, and by replacing macros with their expansion. The first example expands to two different results, depending on whether feature X is defined. The second example is more complex and expands to six different results, depending on whether features X, Y, and Z are selected. Note how macro B has two alternative expansions, depending on the feature selection.

Next to the listings, we show the desired parsing result in form of an abstract syntax tree with variability. The result reflects variability with choice nodes over a feature F (denoted as "$\Diamond_F$"), the left child contains the parse result if feature F is selected, the right branch if it is not selected.[1] By replacing choice nodes with their left or right branch,

---

[1] Abstract syntax trees with variability are known from tools, such as fmp2rsm [14], FeatureMapper [27], and CIDE [31], and from formalisms,

```
1  3 * 7 +
2  #ifdef X
3  1
4  #else
5  0
6  #endif
```

```
1  #define A 3

3  #ifdef X
4  #define B 1
5  #else
6  #define B 2
7  #endif

9  3
10 #ifdef X
11 * 7
12 #endif
13 +
14 #ifdef Y
15 #ifdef Z
16 A
17 #else
18 4
19 #endif
20 * B
21 #else
22 0
23 #endif
```
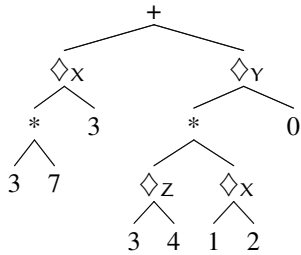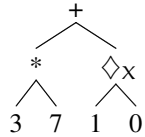
Figure 1. Expressions with conditional fragments and corresponding parse results with choice nodes.


```
1  #define P(msg) \
2    printf(msg);

5  main() {
6    P("Hello\n")
7    P("World\n")
8  }
```

```
1  #ifdef BIGINT
2  #define SIZE 64
3  #endif
4  #ifdef SMALLINT
5  #define SIZE 32
6  #endif

8  allocate(SIZE)
```

```
1  if (!initialized)
2  #ifdef DYNAMIC
3    if (enabled) {
4  #endif
5      init(); //...
6  #ifdef DYNAMIC
7    }
8  #endif
```

Figure 2. Challenges in parsing unpreprocessed code (macro expansion, conditional macros, and undisciplined annotations)

we could perform conditional compilation on the abstract-syntax-tree representation to yield the individual parse results. In many cases however, we want to directly work on a single abstract syntax tree with variability. In our simple expression example, a sample analysis would be an algorithm to approximate upper and lower bounds for the result of numeric expression, which is trivial to implement on the abstract syntax tree with variability, but not on the original source code with preprocessor directives.

2.1 Technical challenges

Parsing unpreprocessed code poses two main technical challenges, which are already visible in our initial expression examples and which we illustrate again on the small C-code snippets in Figure 2.

Macro expansion and file inclusion. Lexical macros and file inclusion interfere with the parsing process. Before parsing we must actually include files and expand macros. In Figure 1, a numeric-expression parser would not understand the token A; in the first C-code example in Figure 2, a

parser would not even recognize two statements because the separating semicolon is added by a macro. Similar to macro expansion, we must resolve includes before parsing, which could contain further macro definitions.

In addition, macros may be defined conditionally (and files may be included conditionally). In the example in Figure 1, macro *B* has two alternative expansions, depending on the feature selection. In the second C-code example, *SIZE* may be expanded in two ways or not expanded at all, depending on the feature selection. To parse unpreprocessed code, we need a mechanism to handle macro expansion and file inclusion (cf. Sec. 4).

Undisciplined annotations. Even without macros and file inclusion, a parser must be able to deal with conditional-compilation directives that do not align with the underlying syntactic structure of the code. The lexical nature of many preprocessors allows developers to annotate individual tokens, like the closing bracket in the third C-code example. Still, the parser must be able to make sense of such annotations and produce a variable abstract syntax tree that is equivalent to the source code in all possible feature combinations.

We call conditional-compilation directives on subtrees of the underlying structure *disciplined annotations* and those that do not align *undisciplined annotations* [32, 40]. In our expression example, most annotations are disciplined: They just provide alternatives for subexpressions. However, the annotation on "*7" (Line 11) is undisciplined since it changes the structure of the resulting abstract syntax tree; in the result, we replicated token 3, so the abstract syntax tree covers both possible structures. Our goal is a parser that can handle undisciplined annotations.

2.2 Soundness, completeness, and performance

Many tool developers have tried to parse unpreprocessed C code with its variability, using different strategies. Soundness, completeness, and performance are three characteristics that we can use to describe desired properties of our parser and to distinguish it from other strategies.

We consider a variability-aware parser as sound and complete if it yields a parse result that correctly represents the variability of the parsed code fragment. That is, it should not matter whether we (a) first generate the source code of one variant (with a standard preprocessor given a desired feature

---

such as the choice calculus [18]. Our contribution lies not in how to encode variability in syntax trees, but in parsing such trees from code.

There are different equivalent abstract syntax trees for the same expression. We can move up the choice nodes in the tree by replicating tokens and we can move down choice nodes by refactoring common children. Two trees are equivalent when they can produce the same variants. Erwig and Walkingshaw [18] have formalized these operations on trees in their choice calculus and have defined normal forms. Some analysis tools may prefer choice nodes at certain levels of granularity only. Moving choice nodes in trees after parsing is straightforward.

selection) and then parse that variant with a standard parser or (b) first parse the unpreprocessed code with a variability-aware parser and then generate the abstract syntax tree of the variant (by pruning subtrees not relevant for the desired feature selection). A variability-aware parser is *incomplete* if it rejects a code fragment even though preprocessing and parsing would succeed for all variants. A variability-aware parser is *unsound* if it produces a parse result which does not correctly represent the result from preprocessing and parsing in all variants (or if it produces a parse result even if at least one variant is not syntax-correct).

Finally, *performance* largely depends on the complexity of the problem an approach is trying to solve. Unfortunately, no sound and complete parsing approach can avoid the inherent complexity of the problem, which is already exponential in the worst case. The challenge is to distinguish inherent complexity (exponential in the worst case, but usually manageable in real-world examples) from accidental complexity induced only by the parsing strategy or tool.

We illustrate soundness, completeness, and performance based on three common strategies to parse unpreprocessed C code (we discuss these approaches in more detail as part of related work in Section 10.1):

- **Brute force.** For many purposes, a simple but effective strategy is to preprocess a file for all (or all relevant) feature combinations in a brute-force fashion and to subsequently parse and analyze the preprocessed variants in isolation. The brute force approach is our benchmark for soundness and completeness. However, it suffers from exponential explosion and quickly becomes infeasible in practice when the number of features grows. Even if the inherent complexity of the problem is low (for example, features affect distinct structures in the same file), the brute force approach parses all feature combinations. Already to parse a file with 20 features, we would need to preprocess and parse the file up to a million times, independent from the actual inherent complexity of the variability in that file.

- **Manual code preparation.** Another common strategy is to support only a subset of possible preprocessor usage. If we give up completeness, for example, by requiring that conditional compilation and macros align with the underlying structure, parsing unpreprocessed code becomes possible in a sound and efficient fashion [6, 42, 68]: The parse results are correct, but we cannot parse all programs. This strategy can only be used on code written according to certain guidelines; for existing code this would require manual rewrites.

- **Heuristics and partial analysis.** Finally, giving up soundness, several researchers have successfully applied heuristics to efficiently parse unpreprocessed C code [23, 24, 46]. They exploit repeating patterns and idioms, such as the common include-guard pattern or capitalized letters for macro names. There are both complete and incomplete heuristic-based parsers. Despite some reported success, unsound heuristics can lead to incorrect parse results and undetected errors, especially in the presence of unusual macro expansions and undisciplined annotations.

Being incomplete or unsound is not a problem per se. Incomplete or unsound approaches (and even the brute-force approach) have been successfully applied for various tasks [6, 23, 47, 48, 64]. However, they do not fit our goals. Since it appears unrealistic to convince developers of projects as large as the Linux kernel to rewrite their code, and considering the vast amount of existing legacy code, we aim for a complete approach that can parse code without preparation. Furthermore, for precise type checking and other error detection, we aim for a sound parsing mechanism without heuristics to avoid both error reports on correct code and undetected errors. In addition, we aim for acceptable performance in real-world settings; although we cannot avoid the inherent complexity, we want to avoid accidental complexity as far as possible.

Conceptually, we design a sound and complete solution in Sections 3–6 and implement a sound and complete parser for Java with conditional compilation; however, due to implementation issues in the lexer, we makes small sacrifices regarding completeness for C, as we will discuss in Section 9.

2.3 A final remark

By no means do we intend to encourage developers to use lexical preprocessors. If they can encode variability with a better mechanism, they should. For example, frameworks and module systems [51], syntactic preprocessors [42, 68], dedicated language constructs for compile-time variability as in the language D, software composition mechanisms such as feature-oriented programming or aspect-oriented programming [2, 35], projectional workbenches [56, 66], and external variability mappings [14, 27, 31] may all have their own shortcomings, but they all do not depend on lexical preprocessing.

Preprocessors have many well-known problems beyond parsing [17, 19, 59] and should have been replaced decades ago. However, we acknowledge that they are still widely used in practice [17, 39] and that they are often the simplest path to introduce variability. We do not recommend using lexical preprocessors, but we intend to support developers who are forced to use them in the vast amount of existing code.

3. Architecture of TypeChef

We consider variability-aware parsing in the larger context of our TypeChef project, which consists of three main components as shown in Figure 3. First, a variability-aware lexer reads the target file (and some configuration parameters) to produce a token stream. The lexer propagates variability from conditional compilation to conditions in the token stream and resolves macros and file inclusion. Second, a variability-aware parser reads the token stream and produces an abstract
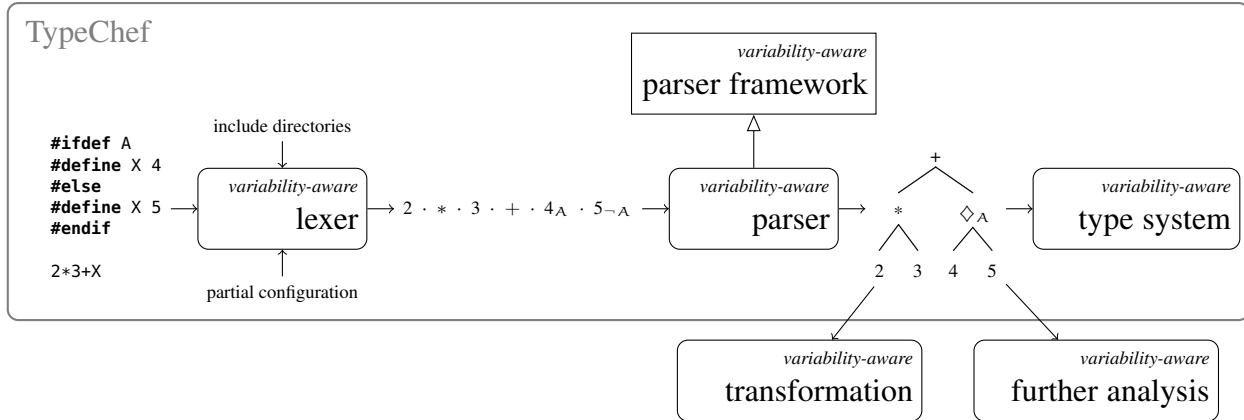
Figure 3. Architecture of the TypeChef project.

syntax tree with variability. Parsers for specific languages are implemented using a generic variability-aware parser framework (a parser-combinator library). Finally, the parse result can be further processed by a variability-aware type system. All components can also be used in isolation and for other purposes.

Our main focus here is the development of the variability-aware parser-combinator library (Sec. 5), the variability-aware parsers for GNU C and Java (Sec. 6), and the application to two case studies (Sec. 7). We already presented the variability-aware lexer, based on earlier ideas of preprocessor analysis [28, 38], in prior work [34]. Nevertheless, we briefly repeat its basic mechanisms (Sec. 4), because it is relevant for understanding how conditional-token streams are produced and how we deal with (conditional) macros.

4. A variability-aware lexer

The variability-aware lexer (formerly also named partial preprocessor) decomposes a code fragment into tokens, propagates variability from conditional compilation into the produced token stream, includes all necessary header files, and expands all macros.

In the produced token stream, each token has a *presence condition*—a propositional formula over features—that evaluates to *true* if the token should be included in compilation.[2] Hence, we speak of *conditional tokens* in a *conditional-token stream*. After a directive #if X (or similar directives for other preprocessors), all tokens receive the presence condition X until the corresponding #endif directive. For nested #if directives, presence conditions are conjuncted ($X \wedge Y$); also #if-#elif-#else-#endif chains are handled accordingly. Deriving presence conditions from #if directives is straightforward, see [57] for a more formal description.

We denote presence conditions as subscripts to tokens, we separate tokens by " · ", and we denote the empty token sequence as "∅". We omit the presence condition $true$ on

tokens that are included in all variants. For example, lexing the first expression from Figure 1 yields the conditional-token stream "$3 \cdot * \cdot 7 \cdot + \cdot 1_A \cdot 0_{\neg A}$."

For preprocessors that provide only conditional compilation, such as Antenna for Java ME, tokenizing and detecting presence condition is sufficient and easy to implement. For the C preprocessor, we resolve also file inclusion and macros.

File inclusion and macros. Handling file inclusion and macros is straightforward in principle. When including a header file, we simply continue reading tokens from that file; when reading a token for which a macro expansion is defined, we return the expansion of the macro. When there are multiple definitions of a macro (e.g., defined in different conditional compilation blocks as in Figures 1 and 2), we return all possible expansions with corresponding presence conditions.[3] Hence, the second expression in Figure 1 results in the following conditional-token stream:

$$3 \cdot *_X \cdot 7_X \cdot + \cdot 1_{Y \wedge Z} \cdot 4_{Y \wedge \neg Z} \cdot *_Y \cdot 1_{Y \wedge X} \cdot 2_{Y \wedge \neg X} \cdot 0_{\neg Y}$$

There are several nontrivial interactions between conditional compilation, macros, and file inclusion and several nontrivial constructs in macros, which are all handled in the variability-aware lexer (except for two minor implementation issues discussed in Sec. 9; the interested reader may try our lexer online at the project's web page). For example, a macro may expand only under some condition, a macro can be used inside an #if expression, a second macro definition replaces previous definition, a macro may be undefined explicitly with #undef, macros are used for include guards, macros can have parameters (and variadic parameters), macros may use stringification, and many more. A detailed description would exceed the scope of this paper, but we refer the interested reader to related publications [34, 38].

---

[2] For how to encode nonboolean features see Sec. 9.

[3] Internally, in contrast to the macro table of an ordinary preprocessor, our variability-aware lexer stores alternative expansions of a macro in a conditional macro table [34]. In that table, each macro expansion has a corresponding presence condition; for example, in Figure 1, *B* expands to 1 if X and to 2 if ¬X.

```
1  static void rt_mutex_init_task(struct task_struct *p) {
2          raw_spin_lock_init(&p->pi_lock);
3  #ifdef CONFIG_RT_MUTEXES
4          plist_head_init_raw(&p->pi_waiters, &p->pi_lock);
5          p->pi_blocked_on = NULL;
6  #endif
7  }
```

↓ macro expansion by variability-aware lexer ↓

```
1  static void rt_mutex_init_task(struct task_struct *p) {
2  #ifdef CONFIG_DEBUG_SPINLOCK
3  do { static struct lock_class_key __key;
        __raw_spin_lock_init((&p->pi_lock), "&p
4  ->pi_lock", &__key); } while (0)
5  #else
6  do { *(&p->pi_lock) = (raw_spinlock_t) { .raw_lock =
7  #ifdef CONFIG_SMP
8  { 0 }
9  #else
10 { }
11 #endif
12 ,
13 #ifdef CONFIG_DEBUG_LOCK_ALLOC
14 .dep_map = { .name = "&p->pi_lock" }
15 #endif
16 }; } while (0)
17 #endif
18 ;
19 #ifdef CONFIG_RT_MUTEXES
20         plist_head_init_raw(&p->pi_waiters, &p->pi_lock);
21         p->pi_blocked_on = ((void *)0);
22 #endif
23 }
```

Figure 4. Excerpt from file kernel/fork.c in Linux illustrates how variability-aware lexing exposes variability from macros in header files.

Note that variability-aware lexing introduces previously hidden variability from macros and header files into the token stream. In Figure 4, we illustrate one concrete excerpt from the Linux kernel, in which variability, depending on several features defined in header files, becomes apparent only by lexing (we serialize the token stream as C code for illustration purpose): "raw_spin_lock_init" in Line 3 is a macro with alternative expansions, the body of which is expanded again. In the worst case, alternative macros could lead to an exponential explosion of the size of the produced token stream, but for common source code the increase is moderate.[4] In our Linux evaluation, on average, partial preprocessing increases input size by 6.4 times compared to the file preprocessed with a minimal configuration.

5. A library of variability-aware parser combinators

After describing how we create conditional-token streams (and eliminate of macros and file inclusion in the process), we focus on parsing these token streams into abstract syntax trees with choice nodes ("◇").

---

[4] We discuss performance optimizations, e.g., preserving sharing to prevent explosion due to iterative replication, elsewhere [26, 34].

Our strategy is to parse the conditional-token stream in a single pass, but split the parser context on conditional tokens and join the parser contexts again to produce choice nodes in the abstract syntax tree. The parser context is a propositional formula (like presence conditions) that describes for which variants the parser is currently responsible. We determine split and join positions by reasoning about the token's presence conditions and the parser context. We split only when necessary and join early to avoid parsing tokens repeatedly. Nevertheless, we might need to parse some tokens multiple times to handle undisciplined annotations, but since we split and join locally, we avoid the accidental complexity of brute-force approaches.

We illustrate the parsing strategy with an example in Figure 5. In this figure, we explain the process in multiple steps and show produced abstract-syntax-tree nodes and the current position of all parser branches and their context.

We have implemented our parser framework as a parser-combinator library [29] in Scala (and a minimal version in Haskell as well). The parser-combinator library implements recursive-descent parsers (also known as top-down, back-tracking, or LL parsers). Although other parsing technologies likely would have been possible and even more efficient, we opted for top-down parser combinators, because they are easy to understand and modify, which was valuable when exploring different design decisions. For explanation, we use concise pattern-matching pseudo code in a functional style.
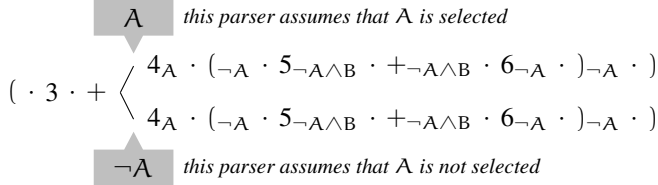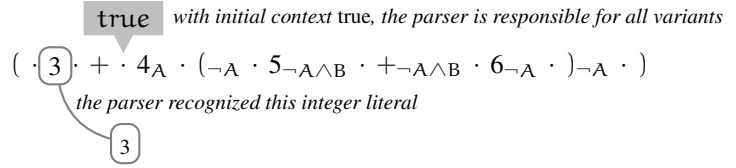
5.1 Parsers and results

A parser in a standard parser-combinator library (e.g., in Scala [44, Ch. 31]) is a function that accepts a token stream and returns a parse result and the remaining token stream, or an error message:

$$\begin{aligned} \mathsf{Parser}[T] \quad &= \mathsf{TokenStream} \to \mathsf{ParseResult}[T] \\ \mathsf{ParseResult}[T] &= \mathrm{Succ}\langle T, \mathsf{TokenStream}\rangle \\ &\quad \mid \mathrm{Fail}\langle Msg\rangle \end{aligned}$$

Two parsers p and q can be combined to form a new parser with the sequence combinator (p~q), the alternatives combinator (p|q), a function-application combinator (p^^f), and others. A standard sequence parser combinator produces a parser that first executes the parser p and, if that does not fail, subsequently executes the parser q on the remaining input stream; the combinator returns the concatenated (tupled) results of both parsers. A standard alternative parser combinator creates a parser that executes parser p and returns either p's result, if it is successful, or the result of calling parser q on the original input. The standard function-application combinator applies a function f to the parse result of p to process the parse result further, for example, to create abstract syntax from parse trees. In that way, semantic actions can be executed while parsing.
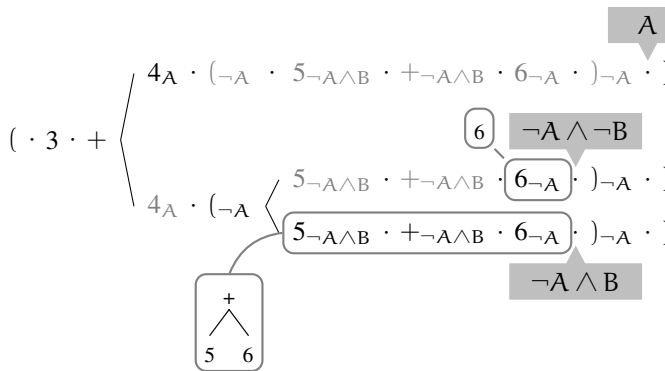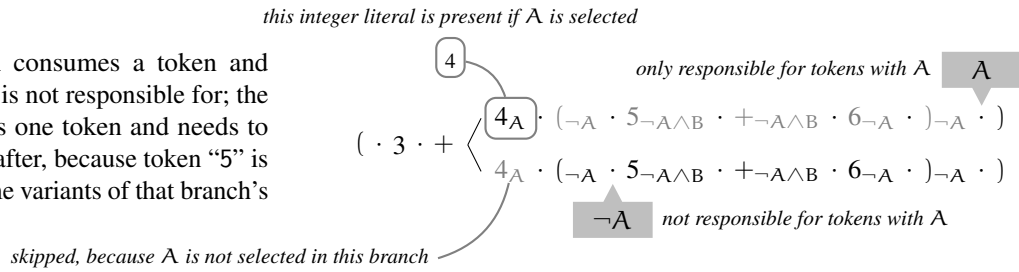
Our variability-aware parsers are similar, but consider a context and multiple possible results (corresponding to

(1) The parser consumes three tokens, but cannot process the fourth token, because that token is not present in all variants.
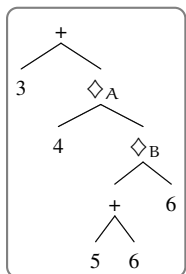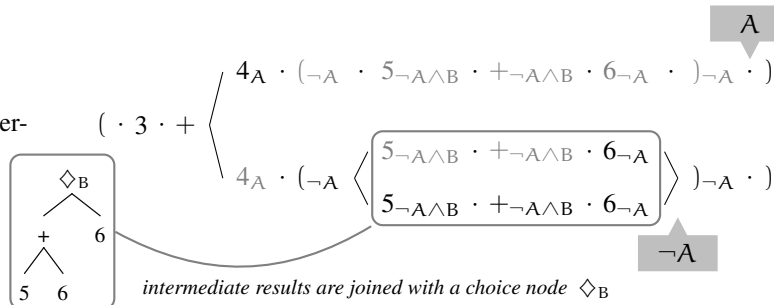
$\texttt{true}$ *with initial context* true*, the parser is responsible for all variants*

$(\ \cdot\ \boxed{3}\ \cdot\ +\ \cdot\ 4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ )$

the parser recognized this integer literal

$\boxed{3}$

A *this parser assumes that* A *is selected*

$(\ \cdot\ 3\ \cdot\ +\ \Big\langle\ \begin{array}{l} 4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ ) \\ 4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ ) \end{array}$

$\neg A$ *this parser assumes that* A *is not selected*

(2) The parser splits into two branches, each with a context responsible for a distinct part of the variant space.

this integer literal is present if A *is selected*

(3) The upper branch consumes a token and skips five tokens it is not responsible for; the lower branch skips one token and needs to split again shortly after, because token "5" is present only in some variants of that branch's responsibility.
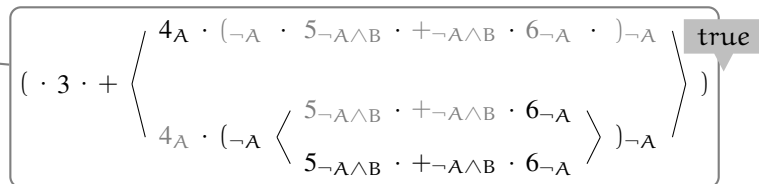
$\boxed{4}$

$(\ \cdot\ 3\ \cdot\ +\ \Big\langle\ \begin{array}{l} \boxed{4_A}\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ ) \\ 4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ ) \end{array}$

only responsible for tokens with A A

$\neg A$ *not responsible for tokens with* A

skipped, because A *is not selected in this branch*

A

$4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ )$

$\boxed{6}$ $\neg A \wedge \neg B$

$(\ \cdot\ 3\ \cdot\ +$

$5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ \boxed{6_{\neg A}}\ \cdot\ )_{\neg A}\ \cdot\ )$

$4_A\ \cdot\ (\neg A$

$\boxed{5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}}\ \cdot\ )_{\neg A}\ \cdot\ )$

$\neg A \wedge B$

$+$
$5\quad 6$

(4) The lower branches both parse an expression and meet at the same position. We need to process token $6_{\neg A}$ twice, because the annotations are not disciplined: $5\ \cdot\ +$ does not form a full expression.

A

$4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ )$

(5) Two parser branches join, producing an intermediate result with a choice node.

$(\ \cdot\ 3\ \cdot\ +$

$\Diamond_B$

$5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}$

$4_A\ \cdot\ (\neg A\ \Big\langle\ \begin{array}{l} 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A} \\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A} \end{array}\Big\rangle\ )_{\neg A}\ \cdot\ )$

$+\quad 6$
$5\quad 6$

$\neg A$

intermediate results are joined with a choice node $\Diamond_B$

$+$
$3\quad \Diamond_A$
$\quad 4\quad \Diamond_B$
$\quad\quad +\quad 6$
$\quad\quad 5\quad 6$

all variability is preserved in the abstract syntax tree

$(\ \cdot\ 3\ \cdot\ +\ \Big\langle\ \begin{array}{l} 4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A} \\ 4_A\ \cdot\ (\neg A\ \Big\langle\ \begin{array}{l} 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A} \\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A} \end{array}\Big\rangle\ )_{\neg A} \end{array}\Big\rangle\ )$

true

(6) The remaining branches join before the final closing bracket, producing another choice node; the parser produces the final result in a single branch.

Figure 5. Example of parsing the conditional token stream $(\ \cdot\ 3\ \cdot\ +\ \cdot\ 4_A\ \cdot\ (\neg A\ \cdot\ 5_{\neg A \wedge B}\ \cdot\ +_{\neg A \wedge B}\ \cdot\ 6_{\neg A}\ \cdot\ )_{\neg A}\ \cdot\ )$.

multiple branches). A parser is a function from a token stream and a context (provided as propositional formula, type Prop) to a result which can either be (a) a parse result with the remaining tokens, (b) an error message, or (c) a split result which contains two inner results depending on some condition (derived from the context and presence condition when splitting). This way, a parser can return multiple results to implement context splits.

$$\begin{aligned}
&\mathsf{VParser}[T] \\
&\quad = \mathsf{Prop} \times \mathsf{TokenStream} \rightarrow \mathsf{VParseResult}[T] \\
&\mathsf{VParseResult}[T] \\
&\quad = \textsc{Succ}\langle T, \mathsf{TokenStream}\rangle \\
&\quad \mid \textsc{Fail}\langle \mathsf{Message}\rangle \\
&\quad \mid \textsc{Split}\langle \mathsf{Prop}, \mathsf{VParseResult}[T], \mathsf{VParseResult}[T]\rangle
\end{aligned}$$

5.2 Context splitting

The fundamental parser on which all further parsers are built, is the parser next which returns the next token. With variability, next may return different results depending on presence conditions and context of the parser. Hence, parser next implements context splitting and is a core mechanism of our variability-aware parser.

Given the parser's context ctx and a token with presence condition pc, there are three possibilities:

1. The parser consumes the token, if and only if it would be present in *all* variants that the parser is responsible for. Technically, that is the case if ctx implies pc (i.e., $ctx \Rightarrow pc$), which we determine with a SAT solver.

2. The parser skips the token, if and only if the parser is not responsible for a single variant in which the token would be present. Technically, that is the case if ctx contradicts pc (i.e., $ctx \Rightarrow \neg pc$).

3. In all other cases, the parser is responsible for some, but not for all, variants in which the token would be present. In this case, we split the parser context into two branches with contexts $ctx \wedge pc$ and $ctx \wedge \neg pc$ (the first of these parsers will consume the token, the second will skip it).

In case we reach the end of the token stream, we return an error message.

$$\begin{aligned}
&\mathsf{next} : \mathsf{VParser}[\mathsf{Token}] \\
&\mathsf{next}(ctx, \emptyset) \\
&\quad = \quad \textsc{Fail}\langle\texttt{"unexpected end of file"}\rangle \\
&\mathsf{next}(ctx, t_{pc} \cdot rest) \\
&\quad = \left\{\begin{array}{ll}
\textsc{Succ}\langle t_{pc}, rest\rangle & \text{if } ctx \text{ implies } pc \\
\mathsf{next}(ctx, rest) & \text{if } ctx \text{ contradicts } pc \\
\textsc{Split}\langle pc, & \text{otherwise} \\
\quad \mathsf{next}(ctx \wedge pc, t_{pc} \cdot rest), \\
\quad \mathsf{next}(ctx \wedge \neg pc, rest)\rangle
\end{array}\right.
\end{aligned}$$

For example, $\mathsf{next}(A, 1_{\neg A} \cdot 2_B \cdot 3)$ skips the first token, because it is never responsible (A contradicts $\neg A$), then splits the result, because it is responsible for some but not for all

variants of the second token (A does not imply B), and returns $\textsc{Split}\langle B, \textsc{Succ}\langle 2, 3\rangle, \textsc{Succ}\langle 3, \emptyset\rangle\rangle$.

Our design is rather unusual, because we reason with a SAT solver during parsing for every token. For each token, we compare parser state and presence conditions to determine split positions and skipping.[5] Although such reasoning can be computational expensive in the worst case (determining tautologies and contradictions is NP-hard), there is evidence that SAT solvers scale well for tasks in variability analysis [33, 43, 62]. We will demonstrate their efficiency also for our parser in our evaluation.

5.3 Filtering

Based on the next parser, we can create more sophisticated parsers, for example, parsers that expect a certain kind of token, such as identifiers, numbers, or closing brackets. For reasoning about variability, these parsers rely entirely on next. Function textToken creates a parser that checks whether the next token has a given textual representation; for example, *textToken '+'* would only accept tokens representing the plus sign. To implement textToken, we use a filter function that checks all successful parse results from next and replaces them by failures in case the expected token does not match.

$$\begin{aligned}
&\mathsf{filter} : (T \rightarrow Bool) \times \mathsf{VParseResult}[T] \rightarrow \mathsf{VParseResult}[T] \\
&\mathsf{filter}(p, \textsc{Succ}\langle val, rest\rangle) \\
&\quad = \left\{\begin{array}{ll}
\textsc{Succ}\langle val, rest\rangle & \text{if } val \text{ satisfies } p \\
\textsc{Fail}\langle\texttt{"unexpected "} + val\rangle & \text{otherwise}
\end{array}\right. \\
&\mathsf{filter}(p, \textsc{Fail}\langle msg\rangle) \\
&\quad = \quad \textsc{Fail}\langle msg\rangle \\
&\mathsf{filter}(p, \textsc{Split}\langle prop, res_1, res_2\rangle) \\
&\quad = \quad \textsc{Split}\langle prop, \mathsf{filter}(p, res_1), \mathsf{filter}(p, res_2)\rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{textToken}(\cdot) : String \rightarrow \mathsf{VParser}[\mathsf{Token}] \\
&\mathsf{textToken}(text)(ctx, input) \\
&\quad = \mathsf{filter}(\lambda t.t \text{ represents } text, \mathsf{next}(ctx, input))
\end{aligned}$$

5.4 Joining contexts

We implement a novel parser combinator for the joining of parse results ($p!$) that attempts to join the results of another parser. The key idea is to move variability out of a split parse result into the resulting abstract syntax tree. That is, instead of multiple parser branches, each with a different context and parse result, we produce a single parser branch with one parse result that contains a choice node (cf. Steps 4–6 in Fig. 5). For

---

[5] In addition, we can also reason about a variability model fm, which describe intended dependencies between features in software product lines, such as feature A requires feature B and mutually excludes feature C. This way, we could restrict the parser only to a subset of variants. Technically, such reasoning is straightforward, we start the parser with context fm instead of $\texttt{true}$ or we consume a token if $fm \Rightarrow ctx \Rightarrow pc$ holds and skip it if $fm \Rightarrow ctx \Rightarrow \neg pc$; the latter has more optimization potential for reasoning with SAT solvers. However, a more detailed discussion is outside the scope of this paper; see [7, 15, 33, 62] for more information about using variability models in variability analysis.

example, join replaces $\text{SPLIT}\langle A, \text{SUCC}\langle 1, 3\rangle, \text{SUCC}\langle 2, 3\rangle\rangle$ by $\text{SUCC}\langle\diamond_A(1, 2), 3\rangle$. Joining early helps to reduce parsing effort for sequencing and alternative parser combinators (see below) and produces smaller abstract syntax trees with choice nodes that are more local.

To join two parser branches in a split parse result, (a) both parser branches must have succeeded and (b) both must, in their respective context, expect the next token *at the same position*. The second condition guarantees that the further behind parser can safely catch up to the position of the other parser, because it would skip the tokens in between anyway. Therefore, after joining, the parser resumes at the position of the further advanced branch as determined with auxiliary function rightmost. If two parse results cannot be merged, join returns the unmodified parse result.

There are different possible encodings of choice nodes in abstract syntax trees. Although easiest in an untyped setting (as typically used for ambiguity nodes in GLR parsing), we want to preserve types in the abstract syntax tree. We therefore wrap a generic Conditional[T] decorator type around each subterm of the abstract syntax tree that should support variability. A Conditional[T] value can either be One if there is no variability or Choice ($\diamond$ for short) in case of alternatives. For example, the abstract syntax tree from the previous example is actually represented as $\text{SUCC}\langle\diamond_A(\text{One}\langle 1\rangle, \text{One}\langle 2\rangle), 3\rangle$. In most examples, however, we omit One for better readability, as already done in previous figures.

$$\begin{aligned}
&\text{Conditional}[T]\\
&\quad = \text{One}\langle T\rangle\\
&\quad \mid \diamond_{\text{Prop}}(\text{Conditional}[T], \text{Conditional}[T])
\end{aligned}$$

$$\begin{aligned}
&\text{join} : \text{Prop} \times \text{VParseResult}[T] \to \text{VParseResult}[\text{Conditional}[T]]\\
&\text{join}(ctx, \text{SUCC}\langle val, rest\rangle) = \text{SUCC}\langle\text{One}\langle val\rangle, rest\rangle\\
&\text{join}(ctx, \text{FAIL}\langle msg\rangle) \quad\quad = \text{FAIL}\langle msg\rangle\\
&\text{join}(ctx, \text{SPLIT}\langle prop, res_1, res_2\rangle)\\
&= \begin{cases}
(\text{SUCC}\langle\diamond_{prop}(val_1, val_2), \text{rightmost}(rest_1, rest_2)\rangle\\
\quad \text{if } res_1' \text{ equals } \text{SUCC}\langle val_1, rest_1\rangle \text{ and}\\
\quad\quad res_2' \text{ equals } \text{SUCC}\langle val_2, rest_2\rangle \text{ and}\\
\quad\quad \text{next}(ctx \wedge prop, rest_1) \text{ equals}\\
\quad\quad\quad \text{next}(ctx \wedge \neg prop, rest_2)\\
\text{SPLIT}\langle prop, res_1', res_2'\rangle\\
\quad \text{otherwise}
\end{cases}\\
&\text{where } res_1' \text{ is join}(ctx \wedge prop, res_1) \text{ and}\\
&\quad\quad res_2' \text{ is join}(ctx \wedge \neg prop, res_2)
\end{aligned}$$

$$\begin{aligned}
&\cdot\,! : \text{VParser}[T] \to \text{VParser}[\text{Conditional}[T]]\\
&(p!)(ctx, input) = \text{join}(ctx, p(ctx, input))
\end{aligned}$$

Optimization. The listed join mechanism attempts to join only parse results that evolved from the same context split. Hence, we might miss some join opportunities. For example, in Figure 6, at the left, we cannot join the branches with results 1 and 3, because they do not share the same parent,

$$\begin{array}{ccc}
\begin{array}{l}
\text{SPLIT}\langle a,\\
\quad \text{SPLIT}\langle b,\\
\quad\quad \text{SUCC}\langle 1, rest_1\rangle,\\
\quad\quad \text{SUCC}\langle 2, rest_2\rangle\rangle,\\
\quad \text{SUCC}\langle 3, rest_1\rangle\rangle
\end{array}
&
\begin{array}{c}
\Rightarrow\\
\text{rewrite}
\end{array}
&
\begin{array}{l}
\text{SPLIT}\langle a \wedge \neg b,\\
\quad \text{SUCC}\langle 2, rest_2\rangle,\\
\quad \text{SPLIT}\langle a,\\
\quad\quad \text{SUCC}\langle 1, rest_1\rangle,\\
\quad\quad \text{SUCC}\langle 3, rest_1\rangle\rangle\rangle
\end{array}
\end{array}$$

Figure 6. Restructuring of a SPLIT tree allows joins across the tree, while preserving variability.

even though they are at the same position $rest_1$ in the token stream. We improve the join mechanism by attempting to join every pair of results. To join two matching nodes at different positions in the SPLIT tree, we simply restructure the tree until the two joinable results are siblings, as illustrated in Figure 6, at the right. For such restructuring, we combine two simple rules: $\text{SPLIT}\langle a, x, y\rangle = \text{SPLIT}\langle\neg a, y, x\rangle$ and $\text{SPLIT}\langle a, \text{SPLIT}\langle b, x, y\rangle, z\rangle = \text{SPLIT}\langle a \wedge b, x, \text{SPLIT}\langle a, y, z\rangle\rangle$. The restructuring preserves the conditions for all results and allows us to apply the original join function subsequently. This improvement allows us to join earlier in some cases; since it is straightforward, but requires more code, we omit a listing.

When to join? By selecting when to join, we can precisely determine where in the resulting abstract syntax tree variability in the form of choice nodes is allowed. Technically, we can attempt to join parse result after every parsing step or after every production. For illustration purposes, we joined at expression level in Figure 1. In our current parser implementation for Java and C, we attempt to join at manually defined positions: in lists, after statements, and after declarations (for details see Sec. 6).

The rationale for this design decision is based on the following trade off: Checking whether split parse results are joinable produces a small computational overhead, hence joining too often may reduce performance. In contrast, joining too late leads to parsing code fragments in split parser branches unnecessarily. In our experience, joining after typical fine-grained program structures seems to be a good balance between computation overhead and low amount of repeated parsing. We have not performed an experimental analysis of how the selection of join positions influences the performance of the parsing algorithm. Note that joining at different positions leads to different (but equivalent) resulting parse trees. For some downstream tools it may be convenient to place choice nodes only at preselected locations in the resulting tree, although rewriting a tree to move choice nodes is always possible as explored in the choice calculus [18].

5.5 Sequencing

The sequence parser combinator ($p{\sim}q$) becomes more complex when alternative results are involved. The produced parser continues all successful results (and only successful results) of the first parser with the second parser. The second

parser might be called multiple times with different contexts (for multiple successful results of the first parser). If the second parser splits the result, the first parse result is copied.[6]

As example, consider parsing the token sequence $1_A \cdot 2_A \cdot 3_{\neg A} \cdot 4$ with next~next. The first parser yields $\text{SPLIT}\langle A, \text{SUCC}\langle 1, 2_A \cdot 3_{\neg A} \cdot 4\rangle, \text{SUCC}\langle 3, 4\rangle\rangle$, and the second parser is called twice, once with context A on $2_A \cdot 3_{\neg A} \cdot 4$ and once with context $\neg A$ on 4. Overall, the parser combinator yields $\text{SPLIT}\langle A, \text{SUCC}\langle 1\sim 2, 3_{\neg A} \cdot 4\rangle, \text{SUCC}\langle 3\sim 4, \emptyset\rangle\rangle$.

Technically, a function seq calls the second parser with the token stream and context of the first parse result (note, the context changes when propagating seq over splitted parse results). Auxiliary function concat simply concatenates (tuples) successful results.

$\text{seq}(ctx, q, \text{SUCC}\langle val_1, rest\rangle) = \text{concat}(val_1, q(ctx, rest))$
$\text{seq}(ctx, q, \text{FAIL}\langle msg\rangle) \quad = \text{FAIL}\langle msg\rangle$
$\text{seq}(ctx, q, \text{SPLIT}\langle prop, res_1, res_2\rangle)$
$\quad = \text{SPLIT}\langle prop, \text{seq}(ctx \wedge prop, q, res_1),$
$\qquad\qquad\qquad \text{seq}(ctx \wedge \neg prop, q, res_2)\rangle$

$\text{concat}(val_1, \text{SUCC}\langle val_2, rest\rangle) = \text{SUCC}\langle val_1 \sim val_2, rest\rangle$
$\text{concat}(val_1, \text{FAIL}\langle msg\rangle) \quad = \text{FAIL}\langle msg\rangle$
$\text{concat}(val_1, \text{SPLIT}\langle prop, res_1, res_2\rangle)$
$\quad = \text{SPLIT}\langle prop, \text{concat}(val_1, res_1), \text{concat}(val_1, res_2)\rangle$

$\cdot \sim \cdot : \text{VParser}[S] \times \text{VParser}[T] \to \text{VParser}[S \sim T]$
$(p \sim q)(ctx, input) = \text{seq}(ctx, q, p(ctx, input))$

5.6 Alternatives

The parser combinator for alternatives $(p \mid q)$ is implemented similarly to sequencing. However, instead of concatenating all successful results, it replaces all failures with the result of the second parser, called with the corresponding context. For example, if parser q returns a split parse result $\text{SPLIT}\langle A, \text{SUCC}\langle 1, 2\rangle, \text{FAIL}\langle "\dots"\rangle\rangle$, the second parser is called with context $\neg A$ to replace the failed result (of course the second parser may fail again or return a split parse result). Again, joining the parser before replacing alternatives can reduce effort, so that the second parser is called less often.

$\text{alt}(ctx, input, q, \text{SUCC}\langle val, rest\rangle) = \text{SUCC}\langle val, rest\rangle$
$\text{alt}(ctx, input, q, \text{FAIL}\langle msg\rangle) \quad = q(ctx, input)$
$\text{alt}(ctx, input, q, \text{SPLIT}\langle prop, res_1, res_2\rangle)$
$\quad = \text{SPLIT}\langle prop, \text{alt}(ctx \wedge prop, input, q, res_1),$
$\qquad\qquad\qquad \text{alt}(ctx \wedge \neg prop, input, q, res_2)\rangle$

$\cdot \mid \cdot : \text{VParser}[T] \times \text{VParser}[T] \to \text{VParser}[T]$
$(p \mid q)(ctx, input) = \text{alt}(ctx, input, q, p(ctx, input))$

5.7 Repetition

Parser combinators for repetition (p^* or p^+) can be constructed with sequencing and alternatives. For example, with a parser combinator e that always returns success without consuming a token, we can implement p^* as $(p \sim p^*) \mid e$. This correctly deals with conditional tokens but has performance problems when parsing long lists with optional entries.

To illustrate the problem, consider the following example: For token sequence $1_A \cdot 2 \cdot 3 \cdot 4$, the parser next* splits at the first token and yields the intermediate result $\text{SPLIT}\langle A, \text{SUCC}\langle 1, 2 \cdot 3 \cdot 4\rangle, \text{SUCC}\langle 2, 3 \cdot 4\rangle\rangle$, after which we cannot join because both parser branches are at different positions in the token stream. The next iteration would yield $\text{SPLIT}\langle A, \text{SUCC}\langle 1\sim 2, 3 \cdot 4\rangle, \text{SUCC}\langle 2\sim 3, 4\rangle\rangle$, which, again, we cannot join. Advancing only the behind-most branch is not possible with our combinators so far. In the worst case, we can only join parser results at the end of the list, which means that all list elements after an optional element will be parsed twice. In C code, this problem is critical, because C files are essentially a long list of top-level declarations, several of which are typically optional.

Therefore, we provide a specialized combinator repOpt(p) for repetition, which returns a list of optional entries (instead of a choice of lists). Each entry in this result list has a presence condition. For example, parsing $1_A \cdot 2 \cdot 3$ with repOpt(next) yields the list $\text{OPT}_A(1)$, $\text{OPT}_{\text{true}}(2)$, $\text{OPT}_{\text{true}}(3)$.[7]

Technically, *repOpt* tries the following strategies: First, it tries to suppress splitting and parses the first list element in isolation. If the parser does not skip any tokens in the process, we can add its result with a corresponding presence condition to the result list (i.e., in the example above, we would recognize 1_A directly as $\text{OPT}_A(1)$ without splitting). With *repOpt*, we avoid excessive splitting and joining within lists. For some input streams, splitting cannot always be avoided; then, *repOpt* parses only in the branch that has consumed the fewest tokens so far, in the hope that branches behind catch up and we can join both branches again early.

Similarly, we constructed performance-optimized parser combinators for comma-separated lists ($p \sim (q \sim p)^*$) and non-empty lists (p^+).

5.8 Function application

Finally, we adjusted the parser combinator for function application ($p \, \hat{}\,\hat{}\, f$), that is used to process parse results (e.g., for constructing abstract syntax out of parse trees). The modification is straightforward: We simply apply the function to all results in a split parse result.

5.9 Parsing effort

As for the variability-aware lexer, worst-case complexity (parsing time and output size) is exponential. Unfortunately, this worst-case complexity cannot be avoided—it is inher-

---

[6] Since we use immutable parse results, a pointer to the same shared data structure is sufficient.

[7] $\text{OPT}_c(v)$ is conceptually equivalent to a choice node with an empty branch $\diamondsuit_c(v, \epsilon)$ and counted as choice node for statistics.

$$3 \cdot * \cdot (_A \cdot 2 \cdot + \cdot 4 \cdot)_A \;\Rightarrow$$
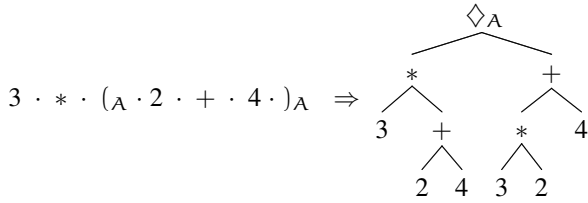
Figure 7. Undisciplined annotation leading to higher parsing effort and replication.

ent in the task—because variability can be used in such a way that for n features there are 2^n completely different outputs. Problematic are code fragments with undisciplined annotations that change the structure of the output, such as the example shown in Figure 7. In addition, we again have to determine satisfiability of propositional formulas (for every combination of presence condition and parser state) which is NP-hard.[8]

Nevertheless, typical source files can be parsed efficiently. There are several characteristics that contribute to efficiency in the common cases:

- We split parser context as late as possible and provide facilities to join parser context early. In contrast to the brute force approach, we avoid accidental complexity by not parsing all tokens before and after a variable segment multiple times.

- If we attempt to join after each parsing step, we guarantee to consume each token only once in token streams with only disciplined annotations (tokens consumed by one parser branch are skipped by all other branches). With disciplined annotations, we do not replicate any nodes in the resulting abstract syntax tree. Only in the presence of undisciplined annotations, we need to consume tokens multiple times and replicate nodes in the result. Although undisciplined annotations are quite typical [40], they mostly occur locally, so that parsing overhead and replication are comparably low.

- Although we need to reason about the relationship between parser context and presence condition for every single token (and once for each parser context), this can be done efficiently with contemporary SAT solvers, even for complex formulas with hundreds of features. We have implemented and fine-tuned a library for propositional formulas (with some extensions for the C preprocessor's facilities for integer constants) in Scala [26] and connected it to the SAT solver *sat4j* [10]. In addition, since typically

---

[8] Already the underlying parsing technology, recursive descent parsing with backtracking, is inefficient. However, we exclude this aspect from discussions within this paper, because we believe that other parsing technologies could be used as well (e.g., packrat parsing [21], LR parsing [25], GLR parsing [63]). We merely used recursive descent parsers because parser combinators made it easy to understand and explore different strategies. In all statistics on consumed tokens, we ignore backtracking-related effort (i.e., a token consumed multiple times in the same context).

```
1  // #ifdef includeMMAPI
2    public void showMediaList(String recordName, ...) { ...
3      // #ifdef includeFavourites
4      if (favorite) {
5        if (medias[i].isFavorite())
6          mediaList.append(medias[i].getMediaLabel(), null);
7      }
8      else
9      // #endif
10       mediaList.append(medias[i].getMediaLabel(), null);
11     ... }
12  // #endif
```

Figure 8. Antenna preprocessor for Java (excerpt from MobileMedia with an undisciplined annotation).

```
1  statement:
2      labeled-statement
3      compound-statement
4      ...
5  labeled-statement:
6      identifier : statement
7      case constant-expression : statement
```

```
1  def statement: MultiParser[Conditional[Statement]] =
2      (labeledStatement | compoundStatement | ...) !
3  def labeledStatement: MultiParser[Statement] =
4      (id ~ COLON ~ statement ^^
5        {case i ~ _ ~ s => LabelStatement(i, s)}) |
6      (textToken("case") ~ constExpr ~ COLON ~ statement ^^
7        {case _ ~ e ~ _ ~ s => CaseStatement(e, s)})
8  def COLON: MultiParser[Token] = textToken(":")
```

Figure 9. Language specification of C and corresponding implementation using our parser combinators (excerpt).

a vast majority of tokens share the same presence conditions, caching is very effective.

We empirically investigate parsing effort in two projects in Section 7.

6. Variability-aware parsers for C and Java

We have used our parser-combinator library to implement parsers for Java 1.5 and GNU C. Although preprocessor usage is less frequent in Java, there are several tools to introduce conditional compilation again. For example, Antenna, often used for variability in Java ME projects for mobile devices, introduces #if and #endif statements in comments as illustrated in Figure 8. For C, we implemented a parser that recognizes C code (specifically the GNU-C dialect used for the Linux kernel) with preprocessor directives of the C preprocessor (again with GNU-specific extensions).

We have implemented both parsers with our parser-combinator framework, which was essentially a straightforward adoption of existing grammars (from JavaCC and ANTLR) and involved some fine-tuning to add missing GNU-C extensions. In Figure 9, we illustrate this implementation with an excerpt from the C language specification [30] and the corresponding Scala implementation using our parser-combinator library (the function-application combinator ^^ is used to create abstract-syntax-tree nodes from token se-

quences; this fragment also illustrates joins at statement level). Although only visible from the type signature, the shown parser fragment is already variability aware by using our specialized parser combinators. As join points for the Java parser, we selected imports, type declarations, modifiers, class members, and statements; for the C parser join points are external definitions, statements, attribute declarations, and type specifiers. We expect that implementing variability-aware parsers for other languages is similarly straightforward.

A technical note on parsing C: To distinguish types from values, a C parser requires a stateful symbol table during parsing. To track the state correctly across parser branches, we implemented a conditional symbol table that tracks under which condition a symbol is declared as type.

7.   Parsing MobileMedia and Linux X86

To evaluate our parser for practical scenarios, we conducted two case studies: We parse the entire unpreprocessed code of (a) the Java ME implementation of MobileMedia and (b) the Linux kernel (X86 architecture). MobileMedia is a favorable case study, due to its small size (5183 nonempty lines of Java code, 14 features) and the absence of macro expansion, so that we can look at parser results without influence of macros in the variability-aware lexer. The Linux kernel is larger by several orders of magnitude—after resolving file inclusion and macros during variability-aware lexing, we parse a total of 899 million nonempty lines of C code (2.6 billion tokens) with 6065 features. At this size, there is no meaningful way to calculate the exact number of valid feature combinations, but the variant space is huge. Parsing Linux required some significant engineering effort to set up and optimize our tools, because of its scale and because variability is additionally managed with the build system (kconfig, kbuild). At the same time, Linux is a good stress test for our tools; for example, we found macro patterns that we never would have expected such as alternative expansions of a macro with different numbers of parameters. From both case studies, we describe the setup, report our experience, and collect statistics on parsing effort.

All scripts and tools used in our case studies are available in the open-source repository of *TypeChef*. We welcome readers to replicate our evaluation and would gladly help to set up parsing other projects.

7.1   Parsing 2400 variants of MobileMedia

MobileMedia is a medium-sized software product line of a Java ME application that manipulates photo, music, and video on mobile devices developed at the University of Lancaster.[9] According to MobileMedia's variability model, we can derive 2400 distinct variants by selecting from 14 features. 36 of 51 files contain variability, typically with 1 to 5 and a maximum of 9 features per file; in total, 14 379 of 23 938 (60 %) tokens are annotated.

Setting up the variability-aware parser is straightforward. Variability-aware lexing is trivial, because no include paths need to be configured and no macros are involved (i.e., files do not grow during lexing). No external configuration knowledge is necessary. We can simply lex and parse each Java file in isolation.

Parsing all 51 files in the entire project takes about 3 seconds.[10] The parser reports all files as syntactically correct in all variants. During parsing, it makes 357 distinct calls to the SAT solver, which requires a negligible time of less than 0.1 seconds in total. The produced parse trees contain a total of 319 choice nodes. Due to undisciplined annotations, such as the one illustrated in Figure 8, we consume 117 of 23 938 tokens twice and a single token three times. That is, variability-aware parsing causes an overhead of only 0.5 % in terms of consumed tokens.

An exact comparison with a brute force strategy is difficult to make without actual preprocessing. However, a rough estimate of a brute force approach per file indicates a parsing overhead of 27 600 %.

7.2   Parsing the Linux kernel with 6065 features

Parsing Linux is more complicated and required some substantial additional engineering effort. In a nutshell, we successfully parsed the entire X86 architecture of Linux release 2.6.33.3 with 6065 variable features and 7665 files (a total of 44 GB, 899 million nonempty lines of C code, and 2.6 billion tokens after variability-aware lexing).[11] Parsing the entire code with our implementation takes roughly 85 hours on a single machine, but is easy to parallelize. For readers interested in details, we describe the setup, practical challenges in the process, and some statistics in the remainder of this section.

Variability implementation in the Linux kernel.   To understand the additional challenges of parsing the Linux kernel, we first describe how Linux is implemented with C, the C preprocessor, and a sophisticated build system. A user can select from over 10 000 features, ranging from different architectures, to different memory models, to numerous drivers, and to various debugging features. Features and their dependencies are described in a variability model, specified in various *Kconfig* files scattered over the source tree [9].

When users want to build a configuration, they invoke a configuration dialog (make config/menuconfig/xconfig) in

---

[9] http://mobilemedia.sf.net/, release 8 OO.

[10] All times in this paper have been measured on normal lab computers (Intel dual/quad-core 3 to 3.4 GHz with 2 to 8 GB RAM; Linux; Java 1.6, OpenJDK). We did not perform low-level optimizations and still compute debug information and statistics. Measured times provide only rough indicators about what performance to expect and that variability-aware parsing is feasible; they are not meant as exact benchmarks.

[11] Without preprocessing the analyzed Linux kernel source is roughly 269 MB; but already ordinary preprocessing increases file size dramatically, because many headers are included in each file. As described in Sec. 4, the output of the variability-aware lexer is roughly 6.4 times larger than the output of an ordinary preprocessor in the minimal configuration.

which they can select the desired features. Most features are of type bool or tristate, that is, they either have two possible values, *include* or *do not include*, or three possible values, *do not include*, *compile into the kernel*, *compile as module*. Few features, such as *Timer frequency* (CONFIG_HZ), have numeric or string values. The configuration mechanism checks and propagates feature dependencies; for example, selecting a feature may deselect dependent features and may prevent to select other features later. The resulting feature selection is written into a configuration file (*.config*).

Variability is implemented both at build system level (deciding which files to compile with which parameters) and at source code level with preprocessor directives (deciding which lines to compile and which macros to expand). Based on the configuration file, the build system decides which files to compile. For each file, the build system can provide alternative or additional directories in which the preprocessor searches for included files (for example, each architecture has a distinct directory for corresponding header files). Files may be compiled and linked differently depending on whether a feature should be compiled as module or as part of the kernel. In few cases, the build system also runs scripts to generate additional files. Finally, the build system passes configured features (as macros prefixed with "CONFIG_") to the preprocessor and C compiler, potentially together with additional definitions from the build script. In the source code, conditional compilation decides which lines to compile and which macros to expand (e.g., #ifdef CONFIG_X queries whether feature X is selected).

Although our variability-aware lexer and parser work without heuristics, the build system is more difficult to analyze. We use analysis tools developed by the research team led by Czarnecki at the University of Waterloo to extract information about features and their dependency from the variability model of the X86 architecture [9, 54, 55] and to extract information about presence conditions on files from the build system [8]. Unfortunately, due to the expressiveness of Linux's variability-modeling language [9], we could only work on a propositional approximation of the full constraints. Worse, the mapping from files to presence conditions is hidden in an imperative build logic within makefiles (using a universal scripting language). To extract those mappings, the analysis tools rely on fuzzy parsing (with unsound heuristics) of the makefiles to recognize patterns. As consequence, for now, we attempt to reduce our dependency on information extracted from the build system and variability model where possible and manually verified involved dependencies in case of reported parsing errors.

A partial configuration. We do not consider all variability, but only 6065 features of the X86 architecture. Specifically, from over 10 000 features in Linux, we deselect all features from other architectures and features that are dead in X86 according to the variability model (i.e., features that may not be selected by a user). The most important practical reason is that the variability-model and build-system extraction tools currently only extract data for the X86 architecture. Furthermore, we exclude 30 features that expect numeric or textual values (we simply use a default; cf. Sec. 9). We also do not consider #ifdef flags without the prefix *"CONFIG_"*, because they are not managed by the Linux variability model—of course the variability-aware lexer handles these flags as well if they are defined or undefined within the source code, we simply assume them not to be defined externally as command line parameters.[12]

Using the information extracted from the build system, we determined which files can be included at all in the considered partial configuration. From that list, we excluded 28 files that depend on files generated by the build system (analyzing the build scripts to generate those files is beyond our scope and using files generated on our system would not reflect the variability available in the corresponding generators; cf. Sec. 9). Of 13 665 C files in all architectures combined, we yield a list with 7665 relevant C files.

Parsing Linux. The parser setup for Linux is straightforward. We iterate over all 7665 C files and run the variability-aware lexer with our partial configuration and the corresponding include paths. We feed the resulting token stream into the parser, together with the presence condition of the entire file extracted from the build system.

The process is *embarrassingly parallel*, that is, trivial to parallelize, since every file can be parsed in isolation from the others (we will need to consider dependencies between files only for type checks, or more accurately linker checks, in future work). Some caching of the results of header files is theoretically possible, but does not seem to be worth the additional effort (given that we would need the exactly same sequence of header files or some nontrivial analysis for sound caching). We simply start the parsing process on multiple machines with a shared disk (usually seven lab computers) and let each computer process the first file not yet started.

Without further setup, the parser reports syntax errors for some feature combinations on many files. We show an excerpt of a typical example and the corresponding error message in Figure 10: If feature CONFIG_SMP is not selected, a header file defines a macro to replace *move_masked_irq* from the source code; however if this macro is expanded in the function definition in *migration.c* it breaks the syntax (by default we report errors with line numbers after macro expansion, because it corresponds to typical debugging tasks; reporting line numbers of the original files is possible as well). After some investigation, we found that the syntax error only occurs in feature combinations that are not allowed by Linux's variability model and cannot be selected when configuring the kernel manually. The file *migration.c* is only parsed when feature CONFIG_GENERIC_PENDING_IRQ is selected,

---

[12] Flags without *"CONFIG_"* are typical for include guards or for compiler-specific variability, such as #ifdef __GNUC__. We defined all flags used by the *gcc* 4.4.5 compiler on our system.

```
                Source: kernel/irq/migration.c            Header: include/linux/irq.h
 1  #include <linux/irq.h>                    240  #ifdef CONFIG_SMP
 2  #include <linux/interrupt.h>              241  ...
                                             242  void move_masked_irq(int
 4  #include "internals.h"                          irq);
                                             243  ...
 6  void move_masked_irq(int                 244  #else /* CONFIG_SMP */
        irq)                                 245  ...
 7  {                                        246  #define move_masked_irq(x)
 8  ...                                      247  #endif /* CONFIG_SMP */

                                                   Code after macro expansion in lexer
277545  ...
277546  void
277547  #if !defined(CONFIG_SMP)

277549  #endif
277550  #if defined(CONFIG_SMP)
277551  move_masked_irq(int irq)
277552  #endif
277553  {
277554  ...

                                                                    Parser output
 1  if CONFIG_SMP:  succeed
 2  if !CONFIG_SMP: failed: end of file expected at line: 277545
```

Figure 10. Conditional parser error in Linux, when not considering presence conditions of files.

which depends on CONFIG_SMP; hence, CONFIG_SMP is always selected when parsing the file and the syntax error cannot occur. After adding a corresponding dependency to the variability model, our parser correctly accepts the file.

Whenever we found a parsing error that cannot occur in practice due to feature dependencies, we added the dependency to an internal model that is used during parsing, leading to the exclusion of the problematic tokens. (We specifically do not use the extracted approximated variability model, because it is not reliable enough.) As a side effect, we are actually reconstructing a small subset of the variability model from parser errors. For parsing the entire X86 architecture, we added 54 such dependencies.

Performance and statistics. Parsing all 7665 files takes approximately 85 hours and correspondingly less when parallelizing the process (for orientation, parsing and compiling a single variant of the kernel requires roughly 15 to 20 minutes with standard tools). On average parsing a file takes 30 seconds, 92 % of all files require less than one minute, only 0.4 % require more than five minutes and the worst case was 22 minutes (caused by many and complex presence conditions in some driver code). Roughly 34 % of all time is spent on lexing and the remaining 66 % on parsing. In general, with a couple of machines, one can run the parser over night to perform some analysis daily, or one can run the parser within minutes on modified files of a change request or commit.

In total, we parsed 2.6 billion tokens with an overhead of 4.1 % (consuming tokens multiple times due to undisciplined annotations). On average, per C file, the variability-aware lexer includes 353 header files, defining 8590 distinct macros (of which 1387 are conditional and 340 have alternative ex-

pansions). After lexing, the average file contains 335 490 tokens, of which 72 % are conditional. We have an average parsing overhead of 4.1 % due to undisciplined annotations. The average file is affected by 207 distinct features (which clearly rules out a brute-force approach) and its tokens have 1779 distinct presence conditions over these features. The average parse result of a file contains 20 097 choice nodes. All averages are described by the median; we show distributions in Figure 11 as box plots.

We could parse all code, which means that we did not find any syntax errors. All syntax errors that we found initially were not actual errors, but were false alarms caused by missing features constraints in the variability model (or by bugs in our tools, which we fixed). For example, we found genuine syntax errors in two dead files (mantis_core.c and i2o_config.c), files which we accidentally parsed due to an inaccuracy in the build-system extraction.

Despite the large code base and huge variants space, the absence of syntax errors is realistic, because we analyzed a released version (and with focus on X86, the most tested architecture) and because Linux developers have a commit process in which changes are carefully reviewed. Compile errors are routinely fixed in the mainline branch before releases. Furthermore, although macros and conditional compilation are sometimes used in extreme ways, the developers mostly follow guidelines how to use macros and conditional compilation in a controlled fashion.[13] We expect problems rather at type-system level than at syntax level;[14] our variability-aware parser lays the foundation for corresponding analysis.

8.   Perspective

Beyond the most obvious use case of detecting syntax errors, there are many further use cases for parsing not only a single variant after preprocessing, but parsing the unpreprocessed code with all variability.

Development support. The parsed abstract syntax trees can be used to enhance integrated development environments. For example, we could provide previews of macro expansion (including all possible combinations for different variants) or extend facilities such as code completion with information about variability. Plenty of editor support for preprocessor-based languages has been proposed [e.g., 37, 65], but, so far, none could rely on a sound abstract syntax tree encoding all variability.

Code transformation. *Refactoring engines* [24, 64] and *transformation systems* [6, 46] typically perform transformations on abstract syntax trees and struggle with variability; for example, a rename-method refactoring usually should rename a method in all variants. Additionally, parsing compile-time

---

[13] cf. */Documentation/SubmittingPatches* in the Linux source.

[14] Type errors occurring only in specific features or feature combinations such as the one reported in http://marc.info/?l=linux-kernel&m=130146346404118&w=2.
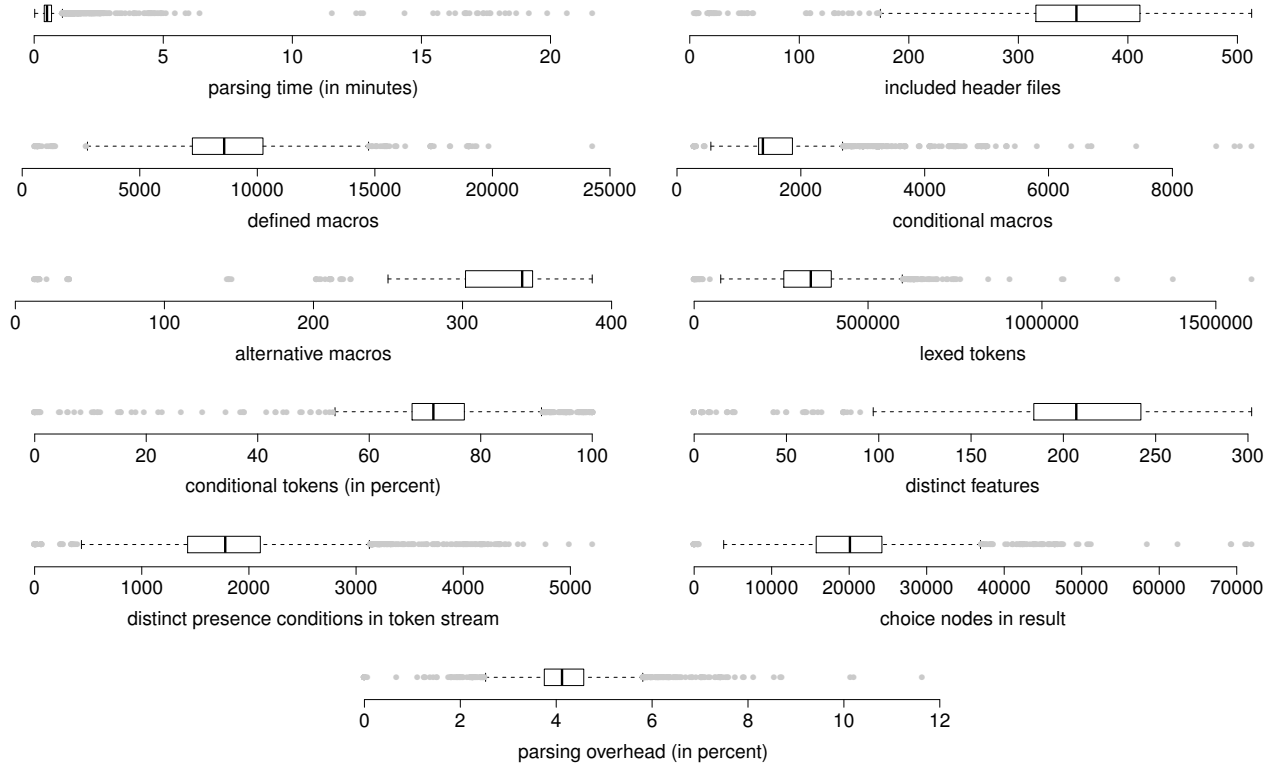
Figure 11. Statistics per file from parsing the Linux kernel (7665 files). Each box plot draws the median as thick line and the quartiles as thin lines, so that 50 % of all measurements are inside the box. The whiskers roughly represent the distribution of remaining values; only a small number of outliers that strongly deviate from the median are drawn as separate dots.

variability allows rewrites of the variability implementation itself. For example, we could rewrite choice nodes into *if* statements depending on global variables and, hence, defer variability decisions from compile time to load time.

Error detection. On the resulting abstract syntax tree with choice nodes, many interesting opportunities for variability-aware analysis arise. A prime candidate is variability-aware type checking, which we outline below. Similarly, extending other existing analyses in a variability-aware fashion, such as *bug finding*, *control-flow analysis*, and *model checking* appears as promising research avenue. Especially for model checking, variability-aware approaches that analyze state systems with feature conditions have already been explored [e.g., 13, 52]; our parser paves the way for translating existing C code with its variability into such state systems.

Variability-aware type checking. The eventual goal of our *TypeChef* project is to type check all variants of the Linux kernel. Without checking each variant in isolation, we want to ensure that all, up to 2^{6065}, variants are well-typed or report corresponding error messages otherwise. Such a type system detects type mismatches, but also dangling function calls. Checks at linker level are a useful extension as well. Variability-aware type systems have been explored before [3, 4, 15, 33, 52, 62], but not on unprepared C code. For example,

we implemented such a type system for Java, working around the parser issue by supporting only disciplined annotations in a controlled environment [33]. With the parsing issue solved, variability-aware type checking for C is in reach.

In a nutshell, the idea is to build a conditional symbol table in which declarations of functions and variables are stored with a corresponding presence condition (much like the conditional macro table of our variability-aware lexer [34]). When we find a function call, we check whether there is one (or more) declaration in scope. We compare the presence condition of call pc^{call} and declarations pc_i^{decl} (presence conditions can be deduced from choice nodes in the abstract syntax tree) and issue an error when there are variants in which the call but not a declaration is present (i.e., if, with variability model VM, $VM \Rightarrow (pc^{call} \Rightarrow \bigvee_i pc_i^{decl})$ does not hold). Similarly, we check that multiple function definitions are mutually exclusive and that types of parameters and return types are compatible in all variants. The type system will provide similar checks for other language constructs. As other variability-aware analyses, the type system directly works on the compact representation of the abstract syntax tree with local variability. Implementing a full variability-aware type system for C is part of our ongoing research.

For the various use cases, the long parse times are of different concern. In many cases, parsing only a single file or few changed files is sufficient (e.g., in editors or analyzing patches). Heavy analysis tasks on a large code base can realistically run in nightly builds with some parallelization.

9. Limitations

Although we were able to show that our setup scales even for the complexity faced in Linux, there are both conceptual and implementation-specific limitations that are important to know to judge the capabilities of our parser. Although we cannot parse arbitrary C code due to these limitations, the parser is *nearly complete* and can handle the vast majority of code fragments as demonstrated with Linux.

We expect that all features are *boolean* and limit presence conditions to *propositional formulas*. We evaluate constraints (such as "#if VER>2") only if the corresponding macros are defined with #define within the source code, but we do not accept numeric constraints over features provided as open command-line parameters. However, we can always (manually or even automatically) encode countable parameters with boolean flags by enumerating all possibilities in the source code (#if VER1 · #define VER 1 · #elif VER2 · #define VER 2 ...). As consequence, we defined 30 nonboolean features with default values as part of our partial configuration of Linux (cf. Sec. 7.2). This limitation of completeness has mainly performance reasons, because we can efficiently reason about propositional formulas with SAT solvers; in principle other solvers would be possible as well.

Variability-aware lexing performs essentially some partial evaluation of macros and includes, which works well because the C preprocessor is simple and not Turing-complete (recursion is limited; preprocessing is guaranteed to terminate; cf. [34]). Lexing is even simpler for languages that do not contain macro expansion or file inclusion, such as *Antenna*. However, there are also more expressive preprocessors that allow arbitrary computations, such as *m4*.[15] To what degree variability-aware lexing is possible for such preprocessors is an open question. Fortunately, handling C preprocessor and simpler forms is sufficient for most practical applications.

A similar problem comes up when considering not only the target language and its preprocessor but also the build system. For example, the Linux build system can run arbitrary scripts and generates some files. So far, we performed only a shallow (and unsound) analysis of the build system and focus on parsing instead (which lead us to exclude 28 files from our evaluation that depend on generated files). This does not affect the soundness of the parser though. Discussions of suitable build systems and how to make them amenable to variability analysis are interesting open research questions, but outside the scope of this paper.

For the variability-aware lexer, we currently do not provide an operation to *undo* macro expansion and file inclusion.

---

[15] http://www.gnu.org/software/m4/

That is unproblematic for our primary goal of type checking, but would be required for refactorings and other source-to-source transformations that should preserve the original code layout. We currently store the original location of tokens for displaying meaningful error messages, but transforming a conditional-token stream back to the original code layout with macros and includes would require additional investigation and nontrivial engineering effort.

Finally, in its current form, the variability-aware lexer is not capable to process some corner-case combinations of conditional compilation with macros using *stringification* [30, §6.10.3]. We manually prepared 13 lines of Linux code (out of 9.5 millions; documented in the repository) to work around these bugs, but we are currently working on a solution. In that regard, the lexer's implementation is not entirely complete, but this is an implementation limitation, not a conceptual one.

10. Related work

Our project touches and combines many areas of research, from parsing unpreprocessed C code, to parsers, to variability implementation (languages and tools), to partial evaluation (in the lexer), to variability-model analysis, to variability-aware type systems, and to several more. For brevity, we discuss only work that is closest to our novel contributions in this paper—parsing unpreprocessed code and practical analysis of Linux. For related work on implementing variability, on the variability-aware lexer, and on variability-aware type systems, see our discussions in prior work [31, 33, 34]; for a comprehensive discussion on variability-model analysis and reasoning about variability, we recommend Benavides' survey [7].

10.1 Parsing of unpreprocessed C code

In Section 2.2, we already introduced three main strategies to parse unpreprocessed C code: brute force, manual code preparation, and heuristics.

The brute-force strategy was used by Vittek to apply refactorings to C code [64]. He simply processed all 2^n combinations of a file separately, where a user has to specify the relevant features manually. While this process may be feasible for the complexity observed in MobileMedia, we argue that it is unrealistic for Linux, except for restricted partial configurations.

Manual code preparation for sound but incomplete parsers was successfully used in projects reported by Baxter and Mehlich [6]. They enforce that conditional compilation directives may only wrap selected syntactic structures (such as entire functions and entire statements; hence preventing constructs with exponential parsing complexity). They extend the C grammar such that the C parser accepts conditional-compilation directives as C language constructs, just like compound statements [6]. Preparing a grammar to understand disciplined annotations is straightforward, adding project-specific patterns (i.e., what is considered disciplined in this

project) is also feasible, but preparing the grammar for all possible uses of preprocessor directives is considered impossible [46]. Baxter and Mehlich report experience that a small team of developers can rewrite an industrial project with 50 000 lines of code "in an afternoon" to make it parseable by this approach. Favre [19] and McCloskey and Brewer [42] provide migration tools to transform lexical preprocessors into disciplined forms (in a different implementation mechanism). However, such migration tools are faced with the same parsing problem and are currently based on unsound heuristics or require human interaction as well. Actually, our parser could be used to make such migration tools more accurate. Nonetheless, we argue that massive code rewrites are unrealistic for projects such as the Linux kernel. Our parser is nearly complete (with the exceptions discussed in Sec. 9) and can parse large code bases without manual preparation.

Finally, good results for parsing unpreprocessed source code at a large scale have been achieved with heuristics [1, 23, 24, 41, 46]. For example, Garrido uses a reversible form of variability-aware lexing (called pseudo preprocessor) together with heuristics to perform refactorings on unpreprocessed C code [23, 24]. Padioleau presents a parser *Yacfe* that accepts most Linux kernel code [46] using a set of heuristics carefully tailored for the project. *Yacfe* does not expand macros, in that regard, the produced abstract syntax tree does not only contain C code, but analysis tools need to understand (or ignore) additional macro nodes. Furthermore, the structure that is recognized is only correct if all assumptions made by the heuristics hold consistently in the entire project; hence, the set of heuristics has to be adapted, fine-tuned, and maintained for each project to parse. Parsing can already go wrong if developers write unusually indented code. Similarly, for an exploratory analysis task, Adams et al. [1] wrote a parser that ignores all code fragments not understood. Badros and Notkin developed a parser PCp^3 that investigates all #ifdef branches, but backtracks and considers only a single path through the document, so alternative macros are not considered [5]. In addition, *srcML* is a tool frequently used to derive code metrics, which tries to roughly recognize code structures and tries to understand preprocessor directives as well (without expanding macros at all) [41]. In our experience with *srcML* [39, 40], we frequently found incorrect results for nontrivial preprocessor usage. We are generally skeptical of heuristics, because it is difficult to judge correctness. In contrast, we used heuristics only for extracting information from the build system, but not for parsing.

Although our parser framework allows to write sound and complete parsers, its performance is orders of magnitude worse compared to solutions based on code preparation or heuristics. For example, *Yacfe* needs 12 minutes to parse the whole Linux kernel [46], compared to 85 hours for X86 in our evaluation. The code's inherent complexity of alternative macro expansions and undisciplined annotations is the root of this performance loss, because we cannot ask developers to rewrite code in a less complex way and we cannot simply ignore complex cases. We still regard the performance of our parser as acceptable for many tasks (usually less than one minute per file, easy to process files in parallel); however, for many tasks faster parsing may outweigh the disadvantages of unsound or incomplete parsing. At the same time, we avoid the accidental complexity of the (sound and complete) brute-force approach to a large degree; parsing the Linux kernel in a brute force fashion, with 90 percent of all files affected by between 124 and 255 distinct features, would be unrealistic.

10.2 Variability-aware parsers

There are a few approaches to parse unpreprocessed C code that are close to our idea of splitting and joining parse results using other parser formalisms. We implement our parser based on parser combinators (the version for eager languages [67]) in Scala [44]. Despite performance drawbacks (backtracking and suboptimal tail-call optimization of Scala in the Java VM), we use LL parsing and a parser-combinator interface because it is easy to understand and allowed us to explore different design decisions. It is possible to integrate our concepts of splitting and joining contexts also with other parser technologies, possibly yielding better performance, and different researchers have worked on similar ideas for LR-based parsers.

First, Platoff et al. sketched a similar parser as part of a general maintenance framework [50]. At *#ifdef* directives, they clone the parser state of an LALR parser and join when both parsers reach an identical state. However, they do not support alternative macros and they do not evaluate how their approach scales beyond medium-sized systems.

Next, Overbey and Johnson outlined a similar strategy, also based on modified LR parsers [45]. They discussed how to handle alternative macros and how to keep a link back to the original source code to allow rewrites. However, they did not fully implement the proposed concepts; currently, their parser *Ludwig* (part of the Photran environment) only processes single configurations without variability.

Finally, in parallel to our work, Gazzillo and Grimm developed *SuperC*, a variability-aware parser based on forking and merging LALR-parser states [25]. In addition to a different parsing technology, they use binary decision diagrams instead of SAT solvers, represent variability in token streams differently, and produce untyped abstract syntax trees. They evaluated performance of their parser using our setup of the Linux kernel (including the information we extracted from the build system) and showed a four times faster performance compared to our parser. However, they do not consider a feature model and did not check for parser errors when parsing Linux (as discussed in Sec. 7.2, without considering dependencies from the feature model, there are many false alarms).

In general, the idea of splitting and joining parser states for variability is similar to GLR parsing, which splits and merges the parser state for ambiguities [63]. GLR parsers return alternative parse results (parse forests) that contain all

matches in case of ambiguities. Technically, they also fork parse stacks similar to our context splits and use a concept called *local ambiguity packing* that is similar to our joins. In contrast to our parser combinators, but more similar to *SuperC*, GLR parsers use sophisticated techniques to advance the parser with multiple contexts synchronously step by step without backtracking.

10.3 Analyzing variability in C code.

There are several studies which investigated variability in C code (often including Linux as case study). However, all studies we are aware of either use unsound heuristics or look only at the preprocessor directives but not at the underlying C code.

In their intention, the works of Tartler et al. are closest to our *TypeChef* project [57, 61]. They analyze #ifdef variability in the Linux kernel to find bugs, currently with a focus of finding code fragments that are never included in any variant. They have reported and confirmed a substantial number of inconsistencies and bugs. However, they perform their analysis entirely at the level of code blocks between preprocessor directives, without considering the underlying code structure. That is, they reason about lines of code and not about abstract syntax trees. At their abstraction level, it is not possible or intended to perform type checking. In addition, they did not consider interactions between macro definition and conditional compilation, as our variability-aware lexer does. In our parser, dead code is simply skipped by all parser branches.

Padioleau's parser *Yacfe* has been used to build code transformations (called semantic patches) and static analysis for Linux [47, 48]. In this line of work, the authors have identified a series of bugs and rule violations (such as "do not use freed memory" or "do not use floating point in the kernel") , however analysis of variability was not in their focus.

At the level of preprocessor directives, several researchers have suggested tools to extract variability information [36] or to provide visualizations, such as #include trees [65]. Some tools also trace macro expansion, but not their interaction with conditional compilation (i.e., neither conditional macros nor alternative macros) [37, 60]. All these approaches intend to support developers in maintenance tasks, but do not analyze the underlying C code in a variability-aware fashion.

Ernst et al. quantified macros and conditional compilation in open-source C code (with a focus on macro usage) [17], and in prior work, also we investigated how preprocessors are used to implement variability [39, 40]. However, those results are based on heuristics (*PCp*[3] and *srcML*, see above) and do not focus on producing abstract syntax trees or detecting errors.

11. Conclusion

We have presented a novel framework for variability-aware parsing, which, together with a variability-aware lexer, can be used to construct parsers that produce abstract syntax trees with variability for unpreprocessed code. Whereas existing approaches suffer either from exponential explosion, require manual code preparation, or are based on unsound heuristics, we have shown that our parser can effectively parse significant code bases without heuristics and code preparation in reasonable time. We have demonstrated practicality on a small Java-based software product line and by parsing the entire X86 architecture of the Linux kernel with 6065 features.

In future work, in our *TypeChef* project, we plan to build a variability-aware type system that can type check the entire Linux kernel and report even type errors hidden in specific feature combinations. Furthermore, performing other variability-aware analysis (e.g., bug patterns, data-flow analysis, model checking) on top of the produced abstract syntax trees is a promising avenue.

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. ACM Press, 2009.

[2] S. Apel and C. Kästner. An overview of feature-oriented software development. *J. Object Technology (JOT)*, 8(5):49–84, 2009.

[3] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer. Type safety for feature-oriented product lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[4] L. Aversano, M. D. Penta, and I. D. Baxter. Handling preprocessor-conditioned declarations. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 83–92. IEEE Computer Society, 2002.

[5] G. J. Badros and D. Notkin. A framework for preprocessor-aware C source code analysis. *Software: Practice and Experience*, 30(8):907–924, 2000.

[6] I. Baxter and M. Mehlich. Preprocessor conditional removal by simple partial evaluation. In *Proc. Working Conf. Reverse Engineering (WCRE)*, pages 281–290. IEEE Computer Society, 2001.

[7] D. Benavides, S. Seguraa, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6):615–636, 2010.

[8] T. Berger, S. She, K. Czarnecki, and A. Wąsowski. Feature-to-code mapping in two large product lines. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 498–499. Springer-Verlag, 2010.

[9] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. Variability modeling in the real: A perspective from the operating systems domain. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 73–82. ACM Press, 2010.

[10] D. L. Berre and A. Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 7(2-3):59–64, 2010.

[11] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004.

[12] BigLevel Software, Inc., Austin, TX. *BigLever Software Gears: User's Guide*, version 5.5.2 edition, 2008.

[13] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 335–344. ACM Press, 2010.

[14] 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)*, volume 3676 of *Lecture Notes in Computer Science*, pages 422–437. Springer-Verlag, 2005.

[15] 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. ACM Press, 2006.

[16] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Trans. Softw. Eng. (TSE)*, 35:573–591, 2009.

[17] M. Ernst, G. Badros, and D. Notkin. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng. (TSE)*, 28(12):1146–1170, 2002.

[18] M. Erwig and E. Walkingshaw. The choice calculus: A representation for software variation. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 21(1), 2011. in press.

[19] J.-M. Favre. Understanding-in-the-large. In *Proc. Int'l Workshop on Program Comprehension*, page 29. IEEE Computer Society, 1997.

[20] E. Figueiredo et al. Evolving software product lines with aspects: An empirical study on design stability. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 261–270. ACM Press, 2008.

[21] B. Ford. Packrat parsing: Simple, powerful, lazy, linear (functional pearl). In *Proc. Int'l Conf. Functional Programming (ICFP)*, pages 36–47. ACM Press, 2002.

[22] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew. Verifying architectural design rules of the flight software product line. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 161–170. ACM Press, 2009.

[23] A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.

[24] A. Garrido and R. Johnson. Analyzing multiple configurations of a C program. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 379–388. IEEE Computer Society, 2005.

[25] P. Gazzillo and R. Grimm. Parsing all of C by taming the preprocessor. Technical Report TR2011-939, Computer Science Department, New York University, 2011.

[26] P. G. Giarrusso. TypeChef: Towards correct variability analysis of unpreprocessed C code for software product lines. Master's

thesis (tesi di diploma di licenza di 2° livello), Scuola Superiore di Catania, 2011.

[27] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping features to models. In *Comp. Int'l Conf. Software Engineering (ICSE)*, pages 943–944. ACM Press, 2008.

[28] 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. IEEE Computer Society, 2000.

[29] G. Hutton and E. Meijer. Monadic parsing in Haskell. *J. Functional Programming*, 8(4):437–444, July 1998.

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

[31] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

[32] C. Kästner, S. Apel, and M. Kuhlemann. A model of refactoring physically and virtually separated features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009.

[33] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type checking annotation-based product lines. *ACM Trans. Softw. Eng. Methodol. (TOSEM)*, 2011. in press.

[34] C. Kästner, P. G. Giarrusso, and K. Ostermann. Partial preprocessing of C code for variability analysis. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 137–140. ACM Press, 2011.

[35] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[36] 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. IEEE Computer Society, 1994.

[37] B. Kullbach and V. Riediger. Folding: An approach to enable program understanding of preprocessed languages. In *Proc. Working Conf. Reverse Engineering (WCRE)*, page 3. IEEE Computer Society, 2001.

[38] M. Latendresse. Rewrite systems for symbolic evaluation of C-like preprocessing. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 165–173. IEEE Computer Society, 2004.

[39] 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. ACM Press, 2010.

[40] J. Liebig, C. Kästner, and S. Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM Press, 2011.

[41] J. I. Maletic, M. L. Collard, and A. Marcus. Source code files as structured documents. In *Proc. Int'l Workshop on Program*

Comprehension (IWPC), page 289. IEEE Computer Society, 2002.

[42] 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. ACM Press, 2005.

[43] 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. ACM Press, 2009.

[44] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima Press, Mountain View, CA, 2008.

[45] J. Overbey and R. Johnson. Generating rewritable abstract syntax trees. In *Proc. Int'l Conf. Software Language Engineering (SLE)*, volume 5452 of *Lecture Notes in Computer Science*, pages 114–133. Springer-Verlag, 2008.

[46] Y. Padioleau. Parsing C/C++ code without pre-processing. In *Proc. Int'l Conf. Compiler Construction (CC)*, pages 109–125. Springer-Verlag, 2009.

[47] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 247–260. ACM Press, 2008.

[48] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proc. Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318. ACM Press, 2011.

[49] 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. IEEE Computer Society, 1997.

[50] M. Platoff, M. Wagner, and J. Camaratta. An integrated program representation and toolkit for the maintenance of C programs. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 192–137. IEEE Computer Society, 1991.

[51] K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin/Heidelberg, 2005.

[52] H. Post and C. Sinz. Configuration lifting: Verification meets software configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350. IEEE Computer Society, 2008.

[53] pure-systems GmbH, Magdeburg. *pure::variants User's Guide*, version 3.0 edition, 2009.

[54] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. Reverse engineering feature models. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 461–470. ACM Press, 2011.

[55] S. She, R. Lotufo, T. Berger, A. Wąsowski, and K. Czarnecki. The variability model of the Linux kernel. In *Proc. Int'l Work-shop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 45–51. University of Duisburg-Essen, 2010.

[56] C. Simonyi. The death of computer languages, the birth of intentional programming. In *NATO Science Committee Conference*, 1995.

[57] J. Sincero, R. Tartler, 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. ACM Press, 2010.

[58] S. S. Somé and T. C. Lethbridge. Parsing minimization when extracting information from code in the presence of conditional compilation. In *Proc. Int'l Workshop on Program Comprehension (IWPC)*, pages 118–125. IEEE Computer Society, 1998.

[59] H. Spencer and G. Collyer. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf.*, pages 185–198. USENIX Association, 1992.

[60] D. Spinellis. Global analysis and transformations in preprocessed languages. *IEEE Trans. Softw. Eng. (TSE)*, 29(11):1019–1030, 2003.

[61] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the Linux 10,000 feature problem. In *Proc. European Conference on Computer Systems (EuroSys)*, pages 47–60. ACM Press, 2011.

[62] 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. ACM Press, 2007.

[63] M. Tomita. An efficient context-free parsing algorithm for natural languages. In *Proc. Int'l Joint Conf. on Artificial Intelligence (IJCAI)*, pages 756–764. Morgan Kaufmann, 1985.

[64] M. Vittek. Refactoring browser with preprocessor. In *Proc. European Conf. on Software Maintenance and Reengineering (CSMR)*, pages 101–110. IEEE Computer Society, 2003.

[65] K. Vo and Y. Chen. Incl: A tool to analyze include files. In *Proc. USENIX Conference*, pages 199–208. USENIX Association, 1992.

[66] M. Voelter. Embedded software development with projectional language workbenches. In *Proc. Int'l Conf. Model Driven Engineering Languages and Systems (MoDELS)*, volume 6395 of *Lecture Notes in Computer Science*, pages 32–46. Springer-Verlag, 2010.

[67] P. Wadler. How to replace failure by a list of successes. In *Proc. Conf. Functional Programming Languages and Computer Architecture (FPCA)*, pages 113–128. Springer-Verlag, 1985.

[68] D. Weise and R. Crew. Programmable syntax macros. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 156–165. ACM Press, 1993.