TypeChef: Toward Type Checking #ifdef Variability in C

Andy Kenner
Metop Research GmbH
Magdeburg, Germany
andy.kenner@metop.de

Christian Kästner
Philipps University Marburg
Marburg, Germany
kaestner@informatik.uni-marburg.de

Steffen Haase, Thomas Leich
Metop Research GmbH
Magdeburg, Germany
haase/leich@metop.de

ABSTRACT

Software product lines have gained momentum as an approach to generate many variants of a program, each tailored to a specific use case, from a common code base. However, the implementation of product lines raises new challenges, as potentially millions of program variants are developed in parallel. In prior work, we and others have developed product-line–aware type systems to detect type errors in a product line, without generating all variants. With *TypeChef*, we build a similar type checker for product lines written in C that implements variability with *#ifdef* directives of the C preprocessor. However, a product-line–aware type system for C is more difficult than expected due to several peculiarities of the preprocessor, including lexical macros and unrestricted use of *#ifdef* directives. In this paper, we describe the problems faced and our progress to solve them with *TypeChef*. Although *TypeChef* is still under development and cannot yet process arbitrary C code, we demonstrate its capabilities so far with a case study: By type checking the open-source web server *Boa* with potentially 2^{110} variants, we found type errors in several variants.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features; D.3.4 [**Programming Languages**]: Processors—*Preprocessors*; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Languages, Reliability, Theory

Keywords

Type system, conditional compilation, C, cpp, #ifdef, partial preprocessor, disciplined annotations

1. INTRODUCTION

Software product line engineering is an efficient means to implement variable software. By selecting from a set of features, a developer can generate different program variants from a common product-line implementation. However, variability comes at a price of increased complexity. Instead of developing and testing a single variant, developers deal with potentially millions of variants in parallel. Already with a few features, we quickly reach a point at which it is no longer possible to compile and run every possible variant in isolation, due to the vast number of variants (up to 2^n variants for n features).

To address this problem, researchers have developed mechanisms that check certain criteria for the entire product line, instead of checking each variant in isolation. This ranges from simple guarantees of syntactic correctness [22], to dead-code detection [34, 35], to type checks and similar referential-consistency checks [1, 2, 8, 16, 21, 36], and to behavioral checks using, among others, model checking [7, 14, 24, 31]. Usually, the idea is to analyze source code before variant generation when it still includes its variability mechanisms; the approaches check implemented variability against the variability model, which describes all valid feature combinations.

Especially, product-line–aware type checking (or reference checking, or safe composition) has shown to scale [8, 20, 36]; type checking an entire software product line with millions of variants is usually as fast as checking a handful of variants in isolation. Product-line–aware type checking has been explored for different variability implementations, most prominently for AHEAD-style feature modules and class refinements [1, 36] and for annotation-based implementations (typically using some form of conditional compilation, also known as negative variability) [2, 8, 16, 21]. However, corresponding type systems were usually targeted at dialects of Java (or Featherweight Java) which limited their applicability to industrial software product lines.

A typical setting to implement variability in industrial software product lines is to use C as programming language [17] and use conditional-compilation directives (*#ifdef*, *#if*, *#elif*, *#else*, *#endif*) of the C preprocessor *cpp* to implement variability (despite broad criticism on the C preprocessor, which is out of scope here). Although we are unaware of any statistics or surveys on industrial product-line implementation, our personal communication with tool providers and developers indicates that actually a majority of industrial software product lines are implemented with the C preprocessor. For example, HP's product line Owen for printer firmware with over 2000 features implements variability entirely with *cpp* [29, 32]; so do many open source programs [25], of which the Linux kernel with over 8000 features is probably the most prominent example [25, 33, 35]. Three more examples of industrial product lines presented at last year's Software Product Line Conference that implement variability at least partially with preprocessors are Danfoss' product line of frequency converters [18], Wikon's product line of remote control systems [30], and NASA's product line of flight control systems [11].

FOSD'10 October 10, 2010, Eindhoven, The Netherlands

Our overall goal is to type check an entire product line written in C and *cpp* with all its variants. To this end, we construct a product-line–aware type checker called TypeChef (type checking *#ifdef*s). Unfortunately, the transfer from a confined research setting in the Java environment to industrial C code turned out much harder than expected. Even though C's type system is rather simple, the C preprocessor *cpp* makes analysis difficult. In a nutshell, *cpp* works on token level; it can be (and is) used at fine granularity and in patterns that are very hard to understand by analysis tools. Additionally, file inclusion and lexical macro substitution (which were not present in previous Java settings) interfere with our analysis, especially when macros are conditionally defined or have alternative expansions. Finally, also presence conditions for code are more complex; *cpp* does not only allow propositional formulas after *#if* directives, but also integer constants (which may be defined, redefined, or undefined during preprocessing) and various operations on them.

In this paper, we describe the problems of type checking a product line implemented with C and *cpp* and describe our solution with TypeChef so far. Specifically, we designed a partial preprocessor to tame *cpp* directives to a disciplined level; and we designed and implemented a type checker that understands *#ifdef* directives and checks them in a way similar to previous product-line–aware type systems. TypeChef is work in progress; we cannot parse arbitrary C code, yet; we do not support alternatives and some manual code preparation is still necessary. Nevertheless, we have already applied *TypeChef* to a small open-source implementation of the Boa web server with 110 features and found several inconsistencies.

In summary, we make the following contributions: (1) We outline the difficulties of product-line–aware type checking for C code. (2) We present and implement an initial solution with TypeChef. (3) In that context, we propose the concept of a partial preprocessor to handle file inclusion and macro substitution. (4) We demonstrate how TypeChef can detect inconsistencies in a small case study.

2. PRODUCT-LINE–AWARE TYPE CHECKING

Let's start by revisiting the basic idea behind product-line–aware type systems, without considering the particularities of *cpp*, yet.

Consider the trivial code fragment in Figure 1. It does nothing more than output a single line of text. However, which text this is depends on preprocessor flags (or features in a product-line context). Lines 4 and 7 are only compiled if the corresponding features are selected; otherwise *cpp* removes the code before compilation. To describe when a code fragment is included, we speak of a *presence condition* $\mathtt{pc}$; a code fragment is only included when its presence condition evaluates to true for the given feature selection. Line 4 has the presence condition $\mathtt{pc}(\mathtt{line4}) = \mathit{WORLD}$, i.e., it is only included when feature *WORLD* is selected.[1] This small program, which we can consider as product line, has two features (*WORLD* and BYE) and can generate four possible variants (with neither feature, with both features, or with either feature). Only two of these four variants will compile though. The compiler will issue an error for the second definition of variable *msg ("Line 7: redefinition of msg")* when both features are selected and will issue an error about a dangling reference *("Line 11: msg undeclared")* when neither feature is selected.

With a product-line–aware type system, we want to guarantee that all potential variants of a product line are well-typed, without

[1]Deriving presence condition from *#ifdef*, *#elif*, and *#else* directives (including nesting) is straightforward; for a formal definition see [34]. A code fragment that is not nested in *#ifdef* directives has the presence condition $\mathtt{true}$, i.e., it is included in all variants.

```
1  #include <stdio.h>
2
3  #ifdef WORLD
4  char * msg = "Hello World\n";
5  #endif
6  #ifdef BYE
7  char * msg = "Bye bye!\n";
8  #endif
9
10 main() {
11   printf(msg);
12 }
```

Figure 1: Example C program

generating all variants. That is, we want to check types before running the preprocessor with a specific feature combination but still guarantee that all variants compile after generation.

In a nutshell, we resolve references and compare annotations in the original code, typically based on an underlying abstract-syntax-tree representation. In Figure 1, *printf* references a function that is declared in the included *stdio.h* file and *msg* references the declarations in Line 4 or 7. Based on these reference pairs, we compare the annotated features. The function call *printf* in Line 11 is not annotated by a feature, neither is the function declaration in *stdio.h*; hence, both call and declaration are included in all variants and do not cause type errors. However, variable *msg* is only declared when feature *WORLD* or feature BYE is included, but referenced in all variants, so we can predict that some variants will not compile. Similarly, we can identify that there are variants in which both variable declarations are included at the same time and issue an error.

In most cases, checking annotations regarding all possible feature combination is too strict though. Typically, domain experts restrict possible feature combinations in a product line, for example, by specifying that either feature *WORLD* or BYE has to be selected in every variant. In product-line engineering, it is best practice to document such domain knowledge in *variability models*. A variability model describes the *intended* variability of the program. Typical forms of variability models are *feature models* and their graphical representation as *feature diagrams* [19], but some projects, such as the Linux kernel, have their own variability-modeling languages [35].

A product-line–aware type system can use a variability model as input and type check only variants allowed by the variability model, instead of all feature combinations. Mathematically, checking only allowed variants is expressed as $\mathit{VM} \rightarrow (\mathtt{pc}(\mathtt{caller}) \rightarrow \mathtt{pc}(\mathtt{target}))$; that is, the presence condition of the caller must imply the presence condition of the target in all variants allowed by the variability model *VM*. Similarly, we can check for redefinitions of variables or functions with $\mathit{VM} \rightarrow \neg(\mathtt{pc}(\mathtt{def1}) \wedge \mathtt{pc}(\mathtt{def2}))$. If the formula is not a tautology (determinable by a SAT solver or other solvers), we issue an error message and can provide an example of a feature selection that causes a type error. For reasoning in a type system, most kinds of variability models can be translated directly into logics [4, 36], and reasoning about them is tractable for even very large models [27]. Our experience shows that the time spent by SAT solvers to determine tautologies is negligible compared to the remaining lookup processes [20].

Given a feature model $\mathit{VM} = (\mathit{WORLD} \vee \mathrm{BYE}) \wedge \neg(\mathit{WORLD} \wedge \mathrm{BYE})$ that defines that exactly one feature must be selected in all variants, we can statically guarantee that the code from Figure 1 is well-typed in all variants: the check regarding *printf* is trivially a tautology ($\mathit{VM} \rightarrow \mathtt{true} \rightarrow \mathtt{true}$), the reference check regarding

msg is a tautology ($VM \rightarrow \mathtt{true} \rightarrow (WORLD \vee BYE)$), and also the condition to prevent redefinition is a tautology ($VM \rightarrow \neg(WORLD \wedge BYE)$).

Checked Properties. Our aim is to find type errors. That is, we want to find the same errors for the entire software product line that a compiler would find for a specific variant. We neither address dynamic properties nor further static properties beyond the type system, such as single assignment. That is, we ensure that each variant compiles, but not that it has meaningful runtime semantics. Adopting static analysis and behavioral checks to software product lines are interesting but separate research challenges [7, 14, 24, 31].

The type system of C is considered as weak, because of implicit type conversion, and unsafe, mainly because of casts between pointers. Chandra and Reps [6] summarize "In C, a pointer of a given type can be *cast* into any other pointer type. Because of this, a programmer can interpret any region of memory to be of any type. Traditional type checking for C cannot enforce that such reinterpretation of memory is done in a meaningful way, because the C standard allows arbitrary type conversions between pointer types. For this reason, C compilers and tools such as *lint* do not provide any warnings against potential runtime errors arising from the use of casts."

Still, there are many kinds of errors that the C type system detects, including dangling variable references (as in Figure 1), dangling function calls, function calls with an incorrect number of parameters, redefinitions of functions and variables, references to undefined types, and type mismatch for assignments and function arguments [17]. Our long-term goal is to cover the entire type system of C as specified in the standard [17]. Nevertheless, we start by checking references (to variables, functions, type declarations), because they are most problematic in a product-line setting, in our experience.

3. PARSING PRE-CPP CODE IS HARD

The main challenge in type checking C code is to parse C code that still contains *cpp* directives (pre-*cpp* code) into a representation that allows us to look up presence conditions and references between elements. Already parsing preprocessed C code is difficult in practice [5], but parsing pre-*cpp* code is a difficult challenge, a challenge already faced by many refactoring and code-analysis tools [3, 9, 10, 13, 26, 28, 37]. Solution strategies either use heuristics [13, 28]—which is not suitable for type checking, since we want to give guarantees—or parse only a subset of possible input programs [3, 26].

A common subset strategy is to build a parser that understands C code with preprocessor directives at certain locations only. For example, *#ifdef* directives may only wrap entire functions or statements, but not arbitrary tokens. We call such restricted use of *#ifdef* directives *disciplined annotations*. Given disciplined annotations, we can create an abstract syntax tree and assign presence conditions to subtrees. Unfortunately, enforcing disciplined annotations may be realistic when writing new code; but an earlier large-scale analysis of 40 cpp-based product lines with a total of 30 million lines of code [20] has shown that on average 11 % of all *#ifdef* directives are *not* in a disciplined form. Consequently, without manual preparation or further tool support, hardly any file can be parsed with this approach.

Additional difficulties come from macro substitution and file inclusion. To parse C code (even when all conditional compilation directives are disciplined), *#include* directives and macros must be expanded. That is, although we want to parse pre-*cpp* code, we need to handle file inclusion and macro substitution during parsing

```
1 #define P(msg) \
2     printf(msg);
3
4 main() {
5   P("Hello\n")
6   P("World\n")
7 }
```

(a) Macro-defined syntactic structure

```
1 #ifdef BIGINT
2 #define SIZE 64
3 #else
4 #define SIZE 32
5 #endif
6
7 allocate(SIZE);
```

(b) Alternative macro expansions

```
1 #ifdef SIZE==64
2 #define BIGINT
3 #endif
4
5 #ifdef BIGINT
6 //...
7 #endif
```

(c) Conditionally defined features

Figure 2: Difficulties in parsing C code

nevertheless. For example, we can only check the reference of the function call *printf* in Figure 1, if we include and parse *stdio.h* first (and recursively the files it includes). To parse the statements in Figure 2a, we need to expand the macro first, which inserts the semicolon necessary to parse the body as two statements. To make matters worse, a macro can have alternative expansions as shown in Figure 2b, and we might need definitions of macros (which may depend on other macros) in future presence conditions as illustrated in Figure 2c.

Parsing pre-*cpp* code is the main challenge for type checking C code, whereas detecting references on an abstract syntax tree and checking presence conditions against a feature model is a straightforward adaptation. Undisciplined annotations, macro substitution, and include directives were all not problems in prior approaches based on Java and its restricted preprocessors or language extensions. In the next section, we describe how we tackle these problems and present a first solution with our tool *TypeChef*.

4. AN OVERVIEW OF TYPECHEF

TypeChef addresses the problem of analyzing pre-*cpp* code in multiple steps—partial preprocessor, expansion to disciplined annotations, parsing, reference analysis, and solving—as illustrated in Figure 3. We discuss each step and its challenges and solutions in isolation.

4.1 Partial Preprocessor

First, we are interested in pre-*cpp* code because of its variability. Nevertheless, we have to expand macros and file inclusions to be able to parse the source code at all. To this end, we contribute a *partial preprocessor*: We pursue the strategy to process macros and file inclusion without affecting variability of conditional compilation constructs. In the example of Figure 3, we recursively include all code from *stdio.h* (for brevity we show only the declaration of method *printf*) and replace all occurrences of the macro *T* by its expansion *char **. Note that the *#ifdef* directives are not changed.

Technically, we currently use a simple hack to implement the partial preprocessor. With a script, we comment out all *#ifdef* directives (except include guards, see below) as illustrated in Figure 4, then run the original preprocessor (which now sees only *#include* and *#define* directives and processes them as usual), and finally remove the comments to restore the *#ifdef* directives. Regarding file inclusion, the preprocessor already provides *#line* directives to maintain information where code came from; this is important to display error messages at the correct location later on. Regarding macro expansion, we do not store information about expansion, yet.

Include guards deserve special attention. An include guard is a standard pattern in C to prevent multiple or recursive inclusions of a file; it uses the same *#ifdef* or *#ifndef* directives as feature code, but follows the pattern illustrated in Figure 5. The partial preprocessor must process include guards, because otherwise indefinite loops can

```
1  #include <stdio.h>
2  #define T char *
3  main(){
4    T msg =
5  #ifdef WORLD
6        "Hello World\n";
7  #else
8        "Bye Bye!\n";
9  #endif
10   printf(msg);
11 }
```

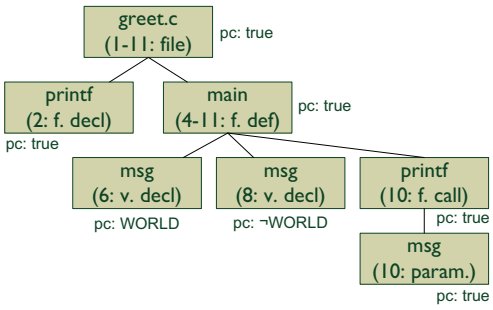
(1) Partial Preprocessor ⇓

```
1  ...
2  int printf(const char *, ...);
3  ...
4  main(){
5    char * msg =
6  #ifdef WORLD
7        "Hello World\n";
8  #else
9        "Bye Bye!\n";
10 #endif
11   printf(msg);
12 }
```

(2) Expansion to Disciplined Annotations ⇓

```
1  ...
2  int printf(const char *, ...);
3  ...
4  main(){
5  #ifdef WORLD
6    char * msg = "Hello World\n";
7  #else
8    char * msg = "Bye Bye!\n";
9  #endif
10   printf(msg);
11 }
```

(3) Parsing ⇓



(4) Reference Analysis ⇓

$$VM \rightarrow (\text{true} \rightarrow \text{true})$$

$$VM \rightarrow (\text{true} \rightarrow (WORLD \vee \neg WORLD))$$

$$VM \rightarrow \neg(WORLD \wedge \neg WORLD)$$

(5) Solving ⇓

"all variants are well-typed"

Figure 3: TypeChef steps

```
1  #include <stdio.h>
2  #define T char *
3  main(){
4    T msg =
5  //#ifdef WORLD
6        "Hello World\n";
7  //#else
8        "Bye Bye!\n";
9  //#endif
10   printf(msg);
11 }
```

Figure 4: Intermediate step of the partial preprocessor: Commenting out conditional compilation directions

```
1  #ifndef _FLAG
2  #define _FLAG
3  ...
4  #endif
```

Figure 5: Include-guard pattern

occur. Hence, we do not comment out preprocessor directives that belong to include guards. Fortunately, we do not need to consider include guards as variability in our type system; it is acceptable to not have them in presence conditions. To distinguish between include guards and *#ifdef* directives that implement variability, we currently use a pattern matching mechanism: *TypeChef* recognizes *#ifndef* and *#define* directives at the beginning of a file and *#endif* at the end of a file as include guard. Alternatively, we could rely on naming conventions, such as "flags for include guards start with an underscore", which are used in most projects anyway.

This simple partial preprocessor, based on comments, works reasonably well. However, it has two limitations, which we currently address in ongoing work. First, and most importantly, we cannot support alternative macro definitions as illustrated in Figure 2b. Second, we cannot use previously defined macros in the condition of an *#if* directive as shown in Figure 2c. In ongoing work, we are developing a more sophisticated partial preprocessor, which can handle alternative macro definitions (by introducing additional *#ifdef* directives at expansion) and conditionally defined feature flags (roughly based on prior work on symbolic execution of *cpp* directives [15, 23]).

4.2 Expansion to Disciplined Annotations

In a second step, we enforce disciplined annotations. As disciplined annotations, *TypeChef* currently allows *#ifdef* directives that wrap one or more entire top-level declarations and definitions (i.e., declarations or definitions of function, structures, unions, and global variables) and directives that wrap one or more statements inside a function, or fields inside a structure or union. In contrast, *TypeChef* considers conditional compilation directives at finer granularity or around partial elements as undisciplined.

In general it is always possible to expand undisciplined annotations to disciplined ones (not all of these expansions are necessarily parseable or well-typed, of course). In the worst case, we can use a brute-force mechanism which replicates the entire code for every possible feature combination. To prevent the exponential complexity, expansions at finer granularity are useful. For example, in Figure 3, we replicate the statement and have two alternative statements instead. In many cases, it might also be possible to manually rewrite to code into a disciplined form, often by introducing additional variables.

```
1  compilation_unit: external_declaration*;
2  external_declaration:
3    function_def |
4    variable_def |
5    '#if' cppexp '\n' external_declaration '\n' cppthenfunc;
6  cppthenfunc:
7    '#endif' '\n' |
8    '#else' '\n' external_declaration '\n' '#endif' '\n' |
9    '#elif' cppexp '\n' external_declaration '\n' cppthenfunc;
10 function_def ...
```

Figure 6: Extended C grammar

Name	Type	Scope	Presence Condition
printf	char * → int	0	true
msg	char *	0	WORLD
msg	char *	0	¬WORLD

Figure 7: Extended symbol table for the example from Fig. 3

Currently, *TypeChef* does not yet automate this step, but a developer has to manually expand undisciplined annotations. In related work, Garridio has implemented such expansion for refactoring C code [12]; we plan a similar tool to automate the task.

4.3 Parsing

Once we have included all files, substituted all macros, and enforced disciplined annotations, the remaining parsing is straightforward. We take a standard C grammar and extend it with productions of *#ifdef* directives as illustrated in Figure 6 (Lines 7–11 are added to detect *#ifdef* directives around top-level declarations). From such grammar, we generate a parser for *TypeChef*, which produces an abstract syntax tree. Parsed *#ifdef* directives are either part of this tree or can be reduced to presence conditions that are annotated at every structural element as shown in Figure 3. For *TypeChef*, we have implemented such parser with the parser generator ANTLR, based on an existing GNU C grammar.

4.4 References Analysis

Based on the abstract syntax tree with presence conditions, TypeChef now looks up references that should be checked. As a result of this step, TypeChef creates a set of formulas (one for each reference or one conjunct formula for all references) that we can later feed into a solver.

Reference lookup in C is mostly straightforward by iterating once over the abstract syntax tree.[2] A simple symbol table, as in Figure 7, is sufficient to store all declared types, variables, and functions, and their respective type, scope,[3] and presence condition. Whenever, we reach a declaration, we add a corresponding entry in the symbol table; in case already an entry with the same name is present (or even multiple), we produce a formula in the form $VM \rightarrow \bigwedge_i \neg(\mathrm{pc}(newDecl) \wedge \mathrm{pc}(prevDecl_i))$ to check that all declarations are mutually exclusive. When we find a function call (or variable access or reference to a type), we look up the function's (variable's, type's) name in the symbol table and retrieve the corresponding presence condition(s). We then produce a corresponding formula $VM \rightarrow (\mathrm{pc}(caller) \rightarrow \bigvee_i \mathrm{pc}(decl_i))$.

[2]Technically, we implemented two iterations, which, however, could be merged.

[3]A scope is necessary for variables to distinguish between variables defined globally or in a function. The distinction is not relevant in our small examples in this paper.

We check references to fields in structures and unions in a similar way; the only difference is that we need to look up the type of a local variable first. That is, right now, we can guarantee that compilation will not fail due to dangling function invocations, and dangling references to variables, fields, or types. In ongoing work, we additionally add checks to ensure consistency between function declarations and functions definitions, and to ensure matching types for assignments, function arguments, and so on; so far, we check only simple references. Also matching signatures in different object files, as checked by the linker, will be addressed.

4.5 Solving

Finally, we need to solve the formulas produced during reference analysis. Throughout this paper, we used propositional formulas for presence conditions and variability models. Actually, the C preprocessor supports more than that: It additionally supports numeric constants and various operations, such as sum, comparison, and bitwise shifting [17]. Therefore, *TypeChef* encodes presence conditions and feature models as constraint satisfaction problem as described by Benavides et al. [4], instead of using a propositional formula. Technically, *TypeChef* uses the constraint-satisfaction-problem solver *Choco*[4] to check for tautologies. We check the formula for each reference in isolation (instead of building one big formula), so that we can trace an error directly to the reference which causes it.

In case a formula is not a tautology, *Choco* finds a counter example representing a specific variant which will not compile. We can present the counter example to the user for further debugging. We can produce an error message that mimics the style of a C compiler (file, line, reason) and that additional provides information about problematic variants.

5. CASE STUDY: BOA WEBSERVER

We implemented *TypeChef* as outlined above. As discussed, *TypeChef* is still work in progress, and there are still significant limitations which prevent applying it to a large-scale industrial C project. Especially, the manual expansion of undisciplined annotations is a severe restriction. Still, we want to demonstrate *TypeChef* at this stage with a (favorable) case study.

As subject of our case study, we selected the open-source web server Boa, version 0.94.13.[5] Boa is a lightweight, single-threaded, and fast implementation of a web server, used mostly in embedded systems and for fast delivery of static content (e.g., `slashdot.org` uses it to deliver image files). It is written in C (6 200 LOC; 38 files) and contains some variability implemented with *cpp*'s *#ifdef* directives. Together, there are 110 different *#ifdef* flags. Some of these flags deal with low-level portability issues, but several can be considered as features in the sense of a product line, for example GUNZIP to support packed HTML files, USE_LOCALTIME to switch between local time and GMT, INET6 to switch between IPv4 and IPv6, three alternative hashing algorithms, and several debug options (logging levels, extra supervision for hash tables, and others). Unfortunately, features and their dependencies are not documented. In theory, there are up to 2^{110} variants of Boa. Even if we consider only some *#ifdef* flags that correspond to end-user variability in a product-line sense (see examples above), we estimate about a thousand possible variants. Hence, generating and checking all variants in isolation does not scale.

We selected Boa because of its manageable size and because almost all of its *#ifdef* directives are in a disciplined form already. Af-

[4]`http://www.emn.fr/z-info/choco-solver/`

[5]`http://www.boa.org/`

```
#ifdef YYPARSE_PARAM
int yyparse (void
     *YYPARSE_PARAM)
#else
int yyparse (void)
#endif
{
  //method body
}
```

(a) Original undisciplined implementation

```
#ifdef YYPARSE_PARAM
int yyparse (void
     *YYPARSE_PARAM)
{
  //method body
}
#else
int yyparse (void)
{
  //method body
}
#endif
```

(b) Expanded disciplined implementation

Figure 8: Alternative method signatures in Boa

ter applying the partial preprocessor, we only needed to expand eight undisciplined annotations, such as the alternative method signatures in Boa's internal (generated) parser shown in Figure 8. Furthermore, neither alternative macros nor conditional feature definitions (cf. Sec. 4.1) cause serious complications in Boa. Boa is a favorable case study that is not significantly affected by the limitations of our current implementation. Nevertheless it is valuable to demonstrate feasibility of our approach and to encourage further improvements toward accepting more and larger C implementations.

With reference analysis, *TypeChef* detects 38 671 references within the entire implementation of Boa (including references within the included header files). These are 2 008 function calls, 7 250 references to variables, 21 934 references to types, and 7 479 references to fields of structures or unions. Of the 38 671 references, 35 478 (92 %) are obviously correct because the target code fragment is not wrapped by *#ifdef* directives or because both elements have the same presence condition. This left us with 3 193 references, which we handed to the solver. Of these, 2 171 (68 %) were tautologies, the remaining 1 022 references are potentially indicators of errors. Additionally, there were 138 potential references for which we did not find a target, which indicate dead or unmaintained code (or incorrect header files in our environment).

We have to be careful with interpreting the solver's results though. To the best of our knowledge, Boa does not have a variability model, neither explicitly nor implicit in some developer documentation. The build environment (*configure* script) does not help either. Nevertheless, domain knowledge that might have been obvious to the original developers might dictate certain dependencies between features, which we were not aware of. Even a single missing dependency in the feature model can lead to many error reports. Finally, there are some false positives caused by limitations of *TypeChef's* current implementation. Hence, we manually inspected the reported errors.

With manual inspection, we could confirm a small number of bugs or undocumented dependencies (we are not familiar enough with the source code to make that judgment). Here, we show two of them with a small code excerpt. First, as illustrated in Figure 9, the flag DEBUG must never be included in any variant, otherwise there will be a dangling reference to *h* in *mmap_cache.c*. Second, as illustrated in Figure 10, the flag HAVE_SYS_FCNTL_H must be included in all variants, otherwise there will be a compilation error due to unknown types in included headers of *alphasort.c*.

TypeChef needs about one minute to check all variants (parsing, reference analysis, and solving), whereas compiling a single variant requires about four seconds on the same system. Solving all equations for the 3 193 non-trivial references takes six seconds with the solver *Choco*. Hence, already with 20 variants, *TypeChef* is faster

```
struct mmap_entry *find_mmap(int data_fd, struct stat *s)
{
    char *m;
    int i, start;
    ...
#ifdef DEBUG
    fprintf(stderr, "New mmap list entry %d (hash was %d)\n",
         i, h);
#endif
```

Figure 9: Detected bug or undocumented dependency in includes of file *mmap_cache.c*

```
#ifdef HAVE_SYS_FCNTL_H
...
  typedef __darwin_off_t off_t;
  typedef __darwin_pid_t pid_t;

...
#endif

int sendfile(int, int, off_t, off_t *, struct sf_hdtr *, int);
...
pid_t fork(void);
```

Figure 10: Detected bug or undocumented dependency in includes of file *alphasort.c*

than the brute-force strategy of compiling all variants in isolation, which is in line with prior experience in product-line–aware type systems for Java [20].

6. RELATED WORK

There have been many approaches to analyze pre-*cpp* code for various purposes. One driving factor were refactorings, which—compared to Java or Smalltalk—are very difficult to implement for pre-*cpp* C code. For example, Vittek used a brute-force-expansion mechanism as sketched in Section 4.2 [37] and Garridio developed a sophisticated mechanism to expand undisciplined annotations at fine granularity [12, 13]. With some heuristics, Garridio's tool was also able to deal with macro expansion and file inclusion, and it could propagate changes back to the original pre-*cpp* code. Similarly, Padioleau uses a sophisticated mechanism based on heuristics (including a significant amount of per-project heuristics) to parse pre-*cpp* code [28]. However, for type checking, we do not want to rely on heuristics; hence, we decided to use a simpler but more accurate mechanism of a partial preprocessor, which is sufficient for type analysis (but would not have been sufficient for refactoring because it cannot propagate changes back).

Additionally, there are several approaches that analyze *cpp* without looking at the underlying code. For example, Tartler et al. analyze C code (including the Linux kernel) for dead code, which cannot be included in any variant [35] and extract presence condition for every code fragment [34]. With a related goal, Hu et al. and Latendresse used symbolic execution to determine presence conditions for all code fragments, also for cases in which features are defined or undefined within the source code as in Figure 2c [15, 23]. Favre extracts the exact semantics of *cpp* for further analysis [10]. These approaches work on lines of arbitrary source code, whereas *TypeChef* looks in between preprocessor directives and analyzes the underlying C code regarding references.

Aversano et al. were the first to suggest to type check a C program including its *#ifdef* variability [2]. They primarily addressed alternative declarations with different types, in contrast to our focus

on references. Their focus was low-level portability of C programs instead of variability in a product line setting, but the solutions and even proposed architectures are similar. In their work, they already proposed an extended symbol table as we used in Figure 7, but, unfortunately, this project was neither implemented nor continued.

The approach to parse pre-*cpp* C code by an extended grammar after manual preparation toward disciplined annotations is often credited to Baxter and Mehlicher [3], who also discussed their experience that 85 % of all *#ifdef* directives are disciplined and the remaining directives received manual attention. In addition, there are several suggestions to replace text-based preprocessors such as *cpp* with a more restricted preprocessor that know the underlying structure [22, 26, 38]. Such approaches restrict conditional compilation constructs to entire language fragments (as we do with disciplined annotations) and either abandon macros or propose syntax macros that are easier to handle. McCloskey and Brewer even provide a semi-automatic migration tool for their disciplined preprocessor *ASTEC* [26]. Unfortunately, we do not expect that we can force developers to switch to a different preprocessor (especially when huge amounts of legacy code are involved) or to manually change their implementations toward disciplined annotations. Therefore, in future work, *TypeChef* aims at preparing the source code automatically during analysis by partial preprocessing and (in the future) automatic expansion of undisciplined annotations.

Finally, in the context of more restricted languages (Java, Featherweight Java, UML), there have been many approaches to check for type errors, reference errors, and other kinds of errors in entire software product lines, e.g., [1, 7, 8, 16, 21, 31, 36]. Their details are beyond the scope of this paper, but the general idea, as outlined in Section 2, is similar in most of them. For a detailed discussion see [20].

7. CONCLUSION

The variability in software product lines provides many opportunities but also complicates development and testing, because a whole family of related variants is developed in parallel. Our goal is to detect implementation errors as early during product-line development and without compiling and testing every variant in isolation. With *TypeChef*, we transfer prior advances in type checking entire software product lines to industrial C code, in which variability is implemented with the C preprocessor *cpp*. Unfortunately, *cpp* has several characteristics that make analysis of unpreprocessed code very difficult. As we have described, to parse C code, we need to expand macros and file inclusion directives and we have to deal with preprocessor directives at every level of granularity and in many undisciplined forms that are difficult to handle.

TypeChef makes first steps toward making C code accessible for product-line–aware type checking. It combines several prior approaches to analyze pre-*cpp* code. With a partial preprocessor, we resolve macros and inclusion directives. With a specialized parser, we can subsequently parse disciplined *#ifdef* directives, analyze references and types within the source code, and detect errors with an off-the-shelf solver. *TypeChef* is work in progress, and in ongoing work, we address limitations, such as alternative macro expansions and undisciplined annotations. Nevertheless, we could already demonstrate the feasibility of *TypeChef* in a favorable case study, which is encouraging for further attempts to type check larger code bases of industrial C code. Our long-term goal is to soundly type check the entire Linux kernel with over 8 000 features and a well-specified variability model.

Acknowledgments. Käster's work is supported by the European Research Council (grant ScalPL #203099).

8. REFERENCES

[1] S. Apel, C. Kästner, Größlinger, and C. Lengauer. Type Safety for Feature-Oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010.

[2] 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. 2002.

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

[4] D. Benavides, P. Trinidad, and A. Ruiz-Cortes. Automated Reasoning on Feature Models. In *Proc. Conf. Advanced Information Systems Engineering (CAiSE)*, pages 491–503. 2005.

[5] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Commun. ACM*, 53(2):66–75, 2010.

[6] S. Chandra and T. Reps. Physical Type Checking for C. In *Proc. Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 66–75. 1999.

[7] 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. 2010.

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

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

[10] J.-M. Favre. CPP Denotational Semantics. In *Proc. Int'l Workshop Source Code Analysis and Manipulation (SCAM)*, pages 22–31. 2003.

[11] 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. 2009.

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

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

[14] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *Proc. Int'l Conf. Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 113–131. 2008.

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

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

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

[18] H. P. Jepsen and D. Beuche. Running a Software Product Line

– Standing Still is Going Backwards. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 101–110. 2009.

[19] K. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, SEI, 1990.

[20] C. Kästner. *Virtual Separation of Concerns*. PhD thesis, University of Magdeburg, 2010.

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

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

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

[24] K. Lauenroth, K. Pohl, and S. Toehning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 269–280. 2009.

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

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

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

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

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

[30] D. Pech, J. Knodel, R. Carbon, C. Schitter, and D. Hein. Variability Management in Small Development Organizations – Experiences and Lessons Learned from a Case Study. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 285–294. 2009.

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

[32] J. G. Refstrup. Adapting to Change: Architecture, Processes and Tools: A Closer Look at HP's Experience in Evolving the Owen Software Product Line. In *Proc. Int'l Software Product Line Conference (SPLC)*, 2009. Keynote presentation.

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

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

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

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

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

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