Type Checking Annotation-Based Product Lines

CHRISTIAN KÄSTNER, Philipps University Marburg
SVEN APEL, University of Passau
THOMAS THÜM and GUNTER SAAKE, University of Magdeburg

Software-product-line engineering is an efficient means to generate a family of program variants for a domain from a single code base. However, because of the potentially high number of possible program variants, it is difficult to test them all and ensure properties like type safety for the entire product line. We present a product-line–aware type system that can type check an entire software product line without generating each variant in isolation. Specifically, we extend the Featherweight Java calculus with feature annotations for product-line development and prove formally that all program variants generated from a well-typed product line are well-typed. Furthermore, we present a solution to the problem of typing mutually exclusive features. We discuss how results from our formalization helped implementing our own product-line tool CIDE for full Java and report of experience with detecting type errors in four existing software-product-line implementations.

1. INTRODUCTION

Software-product-line engineering is an efficient means to create a family of related programs for a domain [Bass et al. 1998; Pohl et al. 2005]. Instead of implementing each program from scratch, product-line engineering facilitates reuse by modeling a domain with *features* (increments in functionality relevant for stakeholders) and generating program *variants* from common assets [Kang et al. 1990; Bass et al. 1998; Czarnecki and Eisenecker 2000]. Hence, from a common code base, we can generate different variants that are tailored to specific usage scenarios. Product-line engineering is typically split into two phases: domain engineering (development of a common code base) and application engineering (variant generation reusing the common code base) [Czarnecki and Eisenecker 2000; Bass et al. 1998; Pohl et al. 2005].

Although the flexibility of software product lines to generate different tailored variants is an important strength [Bass et al. 1998; Pohl et al. 2005], it comes at a price of increased complexity. Instead of a single program, developers implement potentially millions of variants in parallel. To ensure correctness, testing a single program is no longer sufficient; out of millions of variants, errors may occur in few variants that offer a certain feature or feature combination [Pohl et al. 2005;

Thaker et al. 2007; Czarnecki and Pietroszek 2006; Batory and Geraci 1997]. As some variants are never or rarely generated (e.g., only late after initial development when a new customer requests such a variant), potential errors may go undetected for a long time, until they are expensive to fix. A brute force strategy of generating, compiling, and testing all variants is not feasible for most product lines due to the high number of potential variants; therefore, novel approaches are needed that check the entire product line during domain engineering instead of checking each individual variant in isolation during application engineering.

There are many different approaches to implement variability in software product lines. Here, we focus on a simple mechanism, which is very common in industry: Developers *annotate* code fragments inside a common code base, for example, using *#ifdef* statements or similar directives; to generate a variant, annotated code fragments are removed from the common code base, depending on a stakeholder's feature selection. Support for *annotations* (a.k.a. conditional compilation) is available in many environments or languages such as C, C#, Visual Basic, Pascal, Fortran, and Erlang, and Java ME; when not supported natively, it can be added with lightweight tools.

We present a product-line–aware type system that statically and efficiently detects type errors in annotation-based product-line implementations. Type errors are a class of common errors that can be detected statically in many languages, typically during compilation. Product-line implementations are especially prone to type errors, such as dangling method invocations, because variant generation may conditionally remove code. In contrast to conventional product-line approaches that generate and check variants in isolation during application engineering, a product-line–aware type system detects type errors in the entire software product line in a single pass already during domain engineering.

As goals, we want our type system to be both *sound* and *practical*. We formalize the type system for a subset of Java on top of the Featherweight Java calculus [Igarashi et al. 2001] and provide a solution for the problem of type checking alternative (mutually exclusive) features. We *guarantee* that a well-typed software product line produces only well-typed variants (*generation preserves typing*) and prove this property with the proof assistant *Coq*.[1] We deliberately design a backward compatible solution that does not introduce new language constructs but can reuse existing tool infrastructures and can be applied to existing source code. Based on our formalization and gained insights, we implemented a type system for full Java on top of our annotation-based product-line tool CIDE (as extensions to existing type checks in Eclipse). In three case studies, we found that a product-line–aware type system can efficiently detect errors in existing product-line implementations. In all three analyzed product lines, which were developed using *#ifdef* directives by others (between 4 600 and 45 000 lines of code), CIDE found type errors that occur only when a variant with a specific feature combination is generated.

Our type-checking approach is part of a bigger endeavor to detect various kinds of errors in product lines as early as possible. It builds on top of our prior efforts to prevent syntax errors with disciplined annotations in our tool CIDE [Kästner et al. 2008; Kästner et al. 2009] and is inspired by prior work on type checking entire product lines [Czarnecki and Pietroszek 2006; Thaker et al. 2007; Huang et al. 2005; 2007; Delaware et al. 2009; Apel et al. 2010] (see Sec. 9).

This paper is a revised and extended version of [Kästner and Apel 2008]. We make the following contributions (of which only the second was made in [Kästner and Apel 2008]):

— We provide an overview of typing problems and discuss design goals for practical application and reuse of existing tool infrastructures.
— We formalize a product-line–aware type system and a variant generation mechanism on top of Featherweight Java.
— We provide a solution to the problem of typing alternative features.
— We proof soundness (*generation preserves typing*).

---

[1] `http://www.lix.polytechnique.fr/coq/`

— We implement type checks for full Java in CIDE and conduct a series of case studies to evaluate practicality and efficiency of the type system.

2. SOFTWARE-PRODUCT-LINE IMPLEMENTATION

The idea behind software-product-line engineering is to analyze an entire domain and document commonalities and variabilities of different programs of that domain. Then, instead of implementing a single program, developers implement common artifacts from which they can generate different program variants. For example, in the domain of embedded database systems, different program variants are needed depending on different usage scenarios: in some embedded systems transactions are required, in others recovery is needed, others are read-only, and only some need support for ad-hoc queries.

There are many approaches to implement software product lines, ranging from simple ad-hoc mechanisms to sophisticated architectures and specialized languages. In practice, developers often use simple tools such as the C preprocessor to implement variability. In a common code base, developers annotate code fragments with *#ifdef X* and *#endif* directives or similar constructs, in which *X* represents a feature such as transactions. Based on a feature selection provided as configuration file or command line parameters, developers can later include or exclude the annotated code fragments during variant generation.

Beyond languages that support some form of annotations natively, such as C, C#, and Pascal, there are several independent, partly configurable preprocessors such as *GPP*,[2] *GNU M4*,[3] or the preprocessors included in the *Version Editor* [Atkins et al. 2002]. Also the commercial product-line tools *pure::variants* [Beuche et al. 2004] and *Gears* [Krueger 2002] provide their own preprocessors.

In literature, annotation-based approaches are heavily criticized as summarized in the claim "#ifdef considered harmful" [Spencer and Collyer 1992] and in the colloquial term "#ifdef hell" [Lohmann et al. 2006]. Numerous studies discuss the negative effect of preprocessor usage on code quality and maintainability [Spencer and Collyer 1992; Krone and Snelting 1994; Favre 1997; Ernst et al. 2002]. Despite this criticism, practitioners implement many software product lines with preprocessors. Examples are HP's product line Owen for printer firmware [Pearse and Oman 1997], Danfoss' product line of frequency converters [Jepsen and Beuche 2009], the NASA's product line of flight control systems [Ganesan et al. 2009], and the Linux kernel [Tartler et al. 2009; She et al. 2010].

In academia, however, annotation-based approaches received little attention. Instead, academics typically recommend to limit or entirely abandon their use and to implement software product lines with "modern" implementation techniques that encapsulate features in some form of modules (such as components [Szyperski 1997], frameworks [Johnson and Foote 1988], feature modules [Prehofer 1997; Batory et al. 2004], aspects [Kiczales et al. 1997], and others).

Nevertheless, adoption of modern implementation techniques in practice is slow, and we expect that annotation-based product-line implementation will dominate practice at least in the mid-term future. We even discuss that (with some improvement, disciplined usage, and tool support) annotation-based approaches can be considered as viable long-term alternative to module-based approaches [Kästner et al. 2008; Kästner and Apel 2009]; however that discussion is outside the scope of this paper. Here, we provide a type system for annotation-based implementations intended for current and at least mid-term practical use.

3. TYPE ERRORS IN SOFTWARE PRODUCT LINES

Before we start with a formal discussion of our type system, we provide a quick overview of product-line development using annotations, different type errors that can occur, and desirable properties of the type system that we want to guarantee. We provide examples of annotations that result in ill-

---

[2]http://www.nothingisreal.com/gpp/
[3]http://www.gnu.org/software/m4/

typed program variants, which are simplified for conciseness almost to the edge of triviality, but which stem from earlier experience in developing product lines for embedded database applications [Kästner et al. 2007]. Our examples are written in Java and variability is implemented with the well-known syntax of the C preprocessor;[4] however, the same problems occur in other languages and when using other forms of annotations.

Method invocation. As a first example, consider the following code fragment of a class *Storage* used by another class *Database*.

```
1  class Database {
2      void insert(Object key, Object data, Txn txn) {
3          storage.set(key, data, txn.getLock());
4      }
5  }
6  class Storage {
7  #ifdef WRITE
8      boolean set(Object key, Object data, Lock l) {...}
9  #endif
10 }
```

In a read-only database variant, setting values in the storage class is not supported, so the according code is annotated to be removed unless a feature WRITE is selected (*#ifdef*). Although this code is well-typed for all variants that actually select the feature WRITE, the method invocation of *set* in Line 3 (underlined) cannot be resolved in variants in which WRITE is not selected. In these cases, the method invocation remains but the corresponding method declaration is removed. If read-only databases are not generated during development, this error may go undetected. In some cases, it may be detected only after development, when a customer actually requests a variant without WRITE. To type check the entire product line, we need to make sure that the method invocation can *reach* a method declaration in *every* variant in which the invocation itself is present. One of many possible solutions to eliminate the error in our example is to annotate the *insert* method with WRITE as well.

Referencing types. There are numerous similar type errors, for example, when an entire class is annotated as in the example below. If a database without transactions is generated, compilation will fail because the parameter's type *Txn* (underlined) cannot be resolved. Similar type errors can occur when the class is referenced as return type or as supertype, when new objects are instantiated, and in several other cases.

```
1  class Database {
2      void insert(Object key, Object data, Txn txn) {
3          storage.set(key, data, txn.getLock()); }
4  }
5  #ifdef TRANSACTIONS
6  class Txn { ... }
7  #endif
```

Parameters. To fix the previous error, we could annotate the parameter *txn* of the method *insert* as well, as shown below, such that in database variants without transactions *insert* has a different signature. To avoid a problem when accessing the local variable *txn*, we annotate the invocation *'txn.getLock()'*. If a database without transactions is generated, typing this variant still fails, because the method invocation *'storage.set(...)'* has only two parameters, but the method declaration expects three.

---

[4]For the sake of concise examples, we use a slightly relaxed notation of the C preprocessor throughout this paper. First, we allow *#ifdef* instructions inside a line, instead of breaking the source code into multiple lines. Second, we allow boolean operators in the condition as "*#ifdef X ∧ Y*" and "*#ifdef X ∨ Y*" as alternative to nested *#ifdef* directives or "*#if defined(X) || defined(Y)*".

```
1  class Database {
2      void insert(Object key, Object data
3          #ifdef TRANSACTIONS, Txn txn #endif ) {
4          storage.set(key, data #ifdef TRANSACTIONS, txn.getLock() #endif );
5      }
6  }
7  class Storage {
8      boolean set(Object key, Object data, Lock l) {...}
9  }
```

Again, there are different solutions to make all variants in this example well-typed: we can anno-tate the *lock* parameter of *set* (and all occurrences in the method's body not shown here), or we can overload the method declaration of *set*. Either way, when type checking the entire product line, we must make sure that the provided parameters match the expected formal parameters in all variants.

Feature model and alternative features. The previous examples were relatively simple because they contained only annotations with a single optional feature. However, a software product line can have hundreds of features and not all combinations of features may make sense. For example, transactions are not necessary in a read-only database; therefore, we do not need to consider a variant with TRANSACTION but without WRITE during type checking. Furthermore, two features like PERSISTENT and IN-MEMORY for data storage can be alternative (mutually exclusive); every variant must select one of them but not both at the same time. Even more complex constraints like 'feature A can be selected only when B or C but not D are selected' occur in practice [Mendonça et al. 2009; Thüm et al. 2009].

Features and their relationships in product lines are described in a *feature model* (also known as variability model). There are different forms of how to describe feature models; a common form is a feature diagram [Kang et al. 1990; Czarnecki and Eisenecker 2000], but it is also possible to enumerate all allowed variants, or use logics to describe constraints on the feature selection [Batory 2005; Benavides et al. 2005; Schobbens et al. 2006]. Based on a feature model, we can decide which feature combinations are *valid* and can be used to generate a variant. When type checking a software product line, we need to consider all valid variants.

The following code sample shows a code fragment which is only well-typed if we know (a) that PERSISTENT and IN-MEMORY are mutually exclusive (otherwise a variant with both features would be ill-typed because class *Storage* would contain two methods with the same signature) and (b) that WRITE can only be selected if either PERSISTENT or IN-MEMORY is selected (otherwise an ill-typed variant could be generated with a method invocation of *set* but no according declaration). This example illustrates that we need to consider complex constraints between features for type checking product lines.

```
1   class Database {
2   #ifdef WRITE
3       void insert(Object key, Object data, Txn txn) {
4           storage.set(key, data, txn.getLock()); }
5   #endif
6   }
7   class Storage {
8   #ifdef PERSISTENT
9       boolean set(Object key, Object data, Lock lock) {
10          return /* implementation A */;
11      }
12  #endif
13  #ifdef INMEMORY
14      boolean set(Object key, Object data, Lock lock) {
15          return /* implementation B */;
16      }
17  #endif
18  }
```

4. DESIRED PROPERTIES OF THE TYPE SYSTEM

Overall there are two properties that we want to ensure with a type system for software product lines: *generation preserves typing* and *backward compatibility*. The first is the necessary core of this paper and the second is fundamental design decision targeted at better tool support as we will explain.

Generation preserves typing: We want to guarantee that every variant which we can generate from a well-typed product line is well-typed. If a product line allows ill-typed variants, we want an error message upfront during domain engineering, without actually generating every single variant. We call a product line well-typed if all variants it can generate are well-typed. This is the main goal we want to achieve with our type system.

Backward compatibility: We want that a product line that we strip of all its annotations is a well-typed program (not necessarily a variant with reasonable runtime semantics). For our work with Java, this implies two things: (a) our type system is an extension of Java's type system and not a replacement, and (b) we do not introduce new language constructs, because this would be no longer be a Java program. This desired property may appear arbitrary but has a rationale from a tool developer's perspective. As soon as we introduce a new keyword, or just allow multiple methods with the same name as in the previous code example, existing tool infrastructures can no longer be used and must be rewritten. For example, this problem was experienced by the AJDT and Scala teams that provided commercial-quality Eclipse plug-ins for AspectJ and Scala. Because AspectJ and Scala extend the Java syntax, the existing editors with syntax highlighting, outline views, navigation, and code completion could not be reused, but the entire tool infrastructure had to be rewritten (often through 'coping and editing') [Chapman 2006; McDirmid and Odersky 2006]. On the other hand, adopting a new language for product lines without adequate tool support is difficult for developers who are used to the comfort of modern IDEs. Therefore, we design a mechanism and enforce certain restrictions, so that our type system is backward compatible. For example, we do not directly support an implementation as in the previous example, but require a different encoding of alternative features, which we discuss in Section 6.

Backward compatibility is not necessary and can be discussed controversially. On the one hand, if we drop backward compatibility, we can build a more expressive language, especially regarding alternative features. On the other hand, if we retain backward compatibility and design a type system as extension, we can leave the existing type checker and tool infrastructure as is, and just add the additional conditions on top. In fact, in a parallel line of work, we designed a different product-line–aware type system FFJ$_{\mathrm{PL}}$ that drops backward compatibility, introduces new language constructs, and supports alternative features directly [Apel et al. 2010]. The type system is very expressive, but also very complex. Its applicability and ability to scale to realistic product-line implementations has not been shown yet. From our perspective backward compatibility is desirable; it influenced many of our design decisions, which we discuss in the respective sections. We focus on type systems that can be used for industrial-size product-line development and demonstrate the suitability in four case studies in Section 8.

5. COLORED FEATHERWEIGHT JAVA (CFJ)

With *Colored Featherweight Java (CFJ)*, we introduce a calculus of a language and type system for software product lines. We designed CFJ for a subset of Java on top of disciplined annotations. It fulfills both desired properties: generation preserves typing and backward compatibility. (The calculus is named *colored* due to a peculiarity of our product-line tool CIDE, which uses background colors to represent annotations.)

We decided to provide a formalization and proof for both properties, after an initial implementation of our type system for Java. We soon found that our implementation was unsound: We could not give a guarantee and sometimes generated ill-typed variants from a product-line that our implementation had considered well typed, because we forgot some checks. We found similar problems in other product-line–aware type systems (see Sec. 9). At the same time, a formalization of our type

checks for the entire Java language is not feasible because of Java's complexity and rather informal, textual specification (688 pages!) [Gosling et al. 2005]. Instead, we formalize product-line–aware type checking mechanism for *Featherweight Java (FJ)*, a subset of Java, and describe how we implemented and extended it toward full Java and other languages in Section 7.

5.1. Featherweight Java

FJ is a minimal functional subset of Java for which typing and evaluation are specified formally and proved to be sound with the FJ calculus [Igarashi et al. 2001; Pierce 2002]. It was designed to be compact; its syntax, typing judgments and operational semantics fit on a single sheet of paper. FJ strips Java of many advanced features such as interfaces, abstract classes, inner classes, and even assignments, while retaining the core features of Java typing. There is a direct correspondence between FJ and a purely functional core of Java, such that every FJ program is literally an executable Java program.

The motivation behind FJ was to experiment with formal extensions of Java, while focusing only on the core typing features and neglecting many special cases that would require a larger calculus, without raising substantially different typing issues. Because of its simplicity even proofs for significant extensions remain manageable. For the same reasons, we chose FJ over other formalized Java subsets such as Classic Java [Flatt et al. 1998], Java$_{light}$ [Nipkow and von Oheimb 1998], or Lightweight Java [Strniša et al. 2007].

We do not repeat the FJ calculus; however, its mechanisms will become clear from our formalization as we highlight our modifications and repeat unmodified rules.

5.2. Syntax and Annotations

First, we describe CFJ's syntax and how feature annotations are introduced in the calculus. For CFJ, we use the original FJ syntax without casts, as shown in Figure 1.[5] As in FJ, we use the following notational conventions: $\bar{x}$ denotes a list of elements $x_1\ x_2 \ldots x_n$. In conditions of type rules, relations and operations on lists are applied to all entries; for example, $f(\bar{x}) = y$ is short for $\big(f(x_1) = y\big) \wedge \big(f(x_2) = y\big) \wedge \ldots \wedge \big(f(x_n) = y\big)$ and $f(\bar{x}) = g(\bar{y})$ is short for $\big(f(x_1) = g(y_1)\big) \wedge \big(f(x_2) = g(y_2)\big) \wedge \ldots \wedge \big(f(x_n) = g(y_n)\big)$. Finally, also as in FJ, we require elements of lists to be named uniquely; for example, there may not be two methods with the same name in a class.

As in FJ, a class table *CT* maps each class' name to its declaration and has the sanity conditions: (a) $CT(\mathsf{C}) = \mathsf{class}\ \mathsf{C}...$ for every $\mathsf{C} \in dom(CT)$; (b) $\mathsf{Object} \notin dom(CT)$; (c) for every class name C (except Object) appearing anywhere in *CT*, we have $\mathsf{C} \in dom(CT)$; and (d) there are no cycles in the subtype relation (see below) induced by *CT*.

Next, we need to define which code fragments can be annotated and how. There are different ways to model annotations, for example, we could introduce *#ifdef* and *#endif* statements into CFJ's syntax. In fact, the C preprocessor works on plain text without considering the underlying language. Nevertheless, for type checking, we need a higher level of abstraction; we are interested in annotations of code elements such as classes, methods, terms, or parameters. Therefore, we use a different solution: Independent of their actual storage, we provide an external mapping of code elements to features.

In our formalization, we manage annotations using an *annotation table AT* that maps each code fragment to an annotation, similar to the class table *CT* which maps a class name to the corresponding declaration. There are different ways to present annotations to the developer; in the simplest case we can use contemporary preprocessors directives: We parse textual annotations like *#ifdef* of

---

[5]An earlier version of our type system included casts [Kästner and Apel 2008]. Although casts were essential in the original Featherweight Java publication for the discussion about parametric polymorphism [Igarashi et al. 2001], casts do not add anything new for type checking product lines. We decided to remove casts to streamline presentation and proofs.

We make slight modifications to the notation in [Igarashi et al. 2001]: We use $\overline{\mathsf{C}\ \mathsf{f}}$ instead of $\overline{\mathsf{C}}\ \overline{\mathsf{f}}$ to emphasize that it is a list of pairs rather than a pair of lists; the same for $\overline{\mathsf{C}\ \mathsf{x}}$ and $\overline{\mathsf{this}.\mathsf{f}{=}\mathsf{f}}$. Note that $\mathsf{this}.\mathsf{f}{=}\mathsf{f}$ is one syntactic expression and not a relation between two. Additionally, although it is technically not a syntax rule in FJ, we explicitly introduce the program P into the syntax for symmetry in the generation process and proofs later.

P ::=($\overline{\text{L}}$, t)	*program/product line*
L ::=class C extends C { $\overline{\textbf{C f}}$; K $\overline{\textbf{M}}$ }	*class declaration*
K ::=C($\overline{\textbf{C f}}$) { super($\overline{\textbf{f}}$); $\overline{\textbf{this.f=f}}$; }	*constructor decl.*
M::=C m($\overline{\textbf{C x}}$) { return t; }	*method declaration*
t ::=	*terms:*
x	*variable*
t.f	*field access*
t.m($\overline{\textbf{t}}$)	*method invocation*
new C($\overline{\textbf{t}}$)	*object creation*

Fig. 1. CFJ syntax

some surface syntax into the annotation table and remove them from the product line's code base during type checking.

Next, we need to decide what code fragments can be annotated. The C preprocessor can language-independently annotate arbitrary tokens, even just the *class* keyword of a class declaration or its constructor. This makes such preprocessors prone to syntax errors that must be fixed before type checking [Kästner et al. 2009]. Therefore, we map annotations only to code elements that can be removed without invalidating the syntax, in line with our prior work on *disciplined annotations* [Kästner et al. 2008; Kästner et al. 2009]; we simply disallow to annotate in isolation the *class* keyword or other fragments that could cause syntax errors when removed. In CFJ, disciplined annotations are (printed bold in Fig. 1) elements of the class list ($\overline{\text{L}}$), of field and parameter lists ($\overline{\text{C f}}$ and $\overline{\text{C x}}$), method lists ($\overline{\text{M}}$), term lists ($\overline{\text{t}}$), super call parameter lists ($\overline{\text{f}}$), or field assignments ($\overline{\text{this.f=f}}$). When filling the annotation table from a preprocessor, we have to make sure that annotations map only to these code elements and reject all other annotations.

The annotation table is used the following way: $AT(\text{L})$ returns the annotation of a class declaration, $AT(\text{C f})$ returns the annotation of a field, $AT(\text{C x})$ returns the annotation of a parameter, $AT(\text{M})$ returns the annotation of a method, $AT(\text{t})$ returns the annotation of a term, $AT(\text{f})$ and $AT(\text{this.f=f})$ return annotations of parameters and field initializations inside the constructor. Furthermore, we use $AT(\text{C})$ as syntactic sugar for $AT(CT(\text{C}))$ to look up annotations of a class from its name. Note that AT maps annotations from code elements (e.g., identified by their location) to annotations, not from names as CT does. For example, AT can map two methods *foo* in different classes to different annotations; in this case, the result of $AT(\text{foo})$ depends on which declaration of *foo* is referenced. The annotation table is equivalent to introducing annotations into the syntax (which we actually did for our formalization in Coq), but makes the formalization easier to read and is closer to our implementation, in which we avoided to extend the syntax to achieve backward compatibility.

5.3. Reasoning about Annotations

So far, we did not discuss the nature of feature annotations and the feature model. As illustrated in our examples in Section 3, we are interested in reachability conditions like the following: '*whenever code fragment a is present, then also code fragment b is present*'. (We use the metavariables a and b to refer to arbitrary annotatable code fragments.) Reachability is necessary, for example, to check whether a method invocation always references a method declaration, in all variants in that the invocation is present. To determine reachability between code elements a and b, we have to consider the annotations of a and b and the constraints of the feature model. Therefore, we need to define what kind of annotations are possible and how they are evaluated using a feature model.

A feature model describes a set of features and their constraints. A feature selection F is a subset of all features and considered *valid* if the selection fulfills all constraints described in the model. In some formalisms, features can additionally have numeric or textual attributes. For example a feature model may specify that *"feature A is mutually exclusive with feature B and A additionally requires that the attribute x of feature C is larger than 10"*. There are many different ways to describe

feature models, for example, simply enumerating all valid feature combinations, using graphical feature diagrams [Kang et al. 1990; Czarnecki and Eisenecker 2000], or using logics to describe constraints on the feature selection [Batory 2005; Benavides et al. 2005; Schobbens et al. 2006].

Based on features defined in a feature model, different kinds of annotations can decide when to include a code fragment for a feature selection F:

(1) In Thaker's safe composition approach [Thaker et al. 2007], each code fragment is (implicitly) annotated with exactly one feature; a code fragment is removed if the annotated feature is not selected in F.

(2) In our prototype CIDE, by default, each code fragment can be annotated by one feature or a set of features. This is equivalent to *#ifdef* directives and nested *#ifdef* directives of the C preprocessor. For a feature selection F, an annotated code fragment is removed if one of the annotated features is not selected in F.

(3) In *fmp2rsm* [Czarnecki and Pietroszek 2006] and some preprocessors such as *Antenna*, arbitrary propositions such as *'(A or B) and not C'* can be annotated. An annotated code fragment is removed if the formula evaluates to *false* for the feature assignment from F.

(4) Some tools additionally support features with attributes and annotations can reason about attributes (e.g., include code fragment only if text attribute *title* is not *"default"* or if numerical attribute *max-weight* < 10). Examples are the C preprocessor (*#if* directive) and the commercial product-line tool *pure::variants* [Beuche et al. 2004]. Again, the code fragment is removed if the expression evaluates to *false* given a feature selection (with attributes).

In our implementation, we use propositional formulas for feature models and for annotations, but in our formalization, we abstract from concrete formalisms. $AT(\mathsf{a})$ generally returns some expression that evaluates to *false* for a variant with feature selection F (i.e., $eval(AT(\mathsf{a}),F) = false$) when the code fragment a should be removed, while each tool has to provide some (decidable) implementation of *eval*. The empty annotation always evaluates to *true*, thus elements without annotations are never removed. Throughout this paper, we use the term *'a code fragment is present'* for "the code fragment's annotation evaluates to *true*, therefore the element is not removed in the given variant(s)".

We can now define reachability (denoted as $\rightarrow$) between a and b as logical implication in the ordinary sense between $AT(\mathsf{a})$ and $AT(\mathsf{b})$: "whenever $AT(\mathsf{a})$ evaluates to *true* then also $AT(\mathsf{b})$ must evaluate to *true*":

$$AT(\mathsf{a}) \rightarrow AT(\mathsf{b}) ::= \forall F \in \text{valid feature selections}:$$
$$eval(AT(\mathsf{a}),F) \Rightarrow eval(AT(\mathsf{b}),F)$$

In other words, the variants in which code fragment a is included are a subset of (or are the same as) the variants in which code fragment b is included. Bi-implication ($AT(\mathsf{a}) \leftrightarrow AT(\mathsf{b})$) is defined analogously.

A naive approach of determining reachability by iterating over all valid selections does not scale, since there could be millions of valid variants. Still, there are several ways to evaluate the reachability formula efficiently using a SAT solver, a constraint-satisfaction-problem solver, or satisfiability-modulo-theories solvers, depending on how valid feature models, feature selections, and annotations are specified. In the common case that constraints between features can be represented by a propositional formula C_{FM} (e.g., most feature models can be transformed directly into propositional formulas [Batory 2005; Thüm et al. 2009]), and when all annotations can be transformed into propositional formulas (which is possible in most tools), then we can automatically evaluate $AT(\mathsf{a}) \rightarrow AT(\mathsf{b})$ with a SAT solver as described by Thaker et al. [2007]: If the formula $\neg(C_{FM} \Rightarrow (AT(\mathsf{a}) \Rightarrow AT(\mathsf{b})))$ is not satisfiable then b is always reachable from a. For technical details how to reason about feature models and annotations using a SAT solver, see [Batory 2005; Thaker et al. 2007]. As Mendonça et al. [2009] and Thüm et al. [2009] have shown, reasoning about feature models with SAT solvers is tractable for even very large feature models.

5.4. Annotation Rules

Before we model annotation checks formally as extensions in CFJ's typing judgments and prove them complete, we first introduce informally the annotation rules that are to be checked. In general, we need to check code fragments that *reference* other code fragments. The code fragments – references and targets – must be annotated such that the target is *always reachable* from the reference. Otherwise, dangling references that typically result in ill-typed programs can occur. We have identified checks for thirteen different pairs of references and targets:[6]

(L.1) A *class* L can extend only a class that is reachable.

(L.2) A *field* C f can have only a type C of a class L that is reachable.

(K.1) A *super constructor call (i)* can pass only those parameters that are bound to constructor parameters and *(ii)* must pass exactly the parameters expected by the super constructor.

(K.2) A *field assignment* this.f=f in a constructor can *(i)* access only present fields C f in the same class and *(ii)* assign only values that are bound to constructor parameters.

(K.3) A *constructor parameter* C f can have only a type C of a class L that is reachable.

(M.1) A *method declaration* C m(C̄ x) { return t; } can have only a return type C of a class L that is reachable.

(M.2) A method declaration *overriding* another method declaration must have the same signature in all variants in which both are present.

(M.3) A *method declaration parameter* C x can have only a type C of a class L that is reachable.

(T.1) A *variable* x must be bound to a reachable parameter C x of its enclosing method.

(T.2) A *field access* t.f can access only a field C f that is reachable in the enclosing class or its superclasses.

(T.3) A *method invocation* t.m(t̄) *(i)* can invoke only a method M that is reachable and *(ii)* must pass exactly the parameters t̄ expected by this method.

(T.4) An *object creation* new C(t̄) *(i)* can create only objects from a class L that is reachable and *(ii)* must pass exactly the parameters t̄ expected by the target's constructor.

Furthermore, there are some rules that deal with the removal process of children from their parent element. For example, if a class is removed also all methods therein must be removed, if a method is removed also its parameters and its term must be removed. These rules seem obvious and are actually enforced in *#ifdef*-like preprocessors by nesting annotations. However, when formalizing the calculus with arbitrary annotations, we either have to always take all parent annotations into considerations, or we have to make these rules explicit for all elements that can be annotated. We decide for the latter because it provides more flexibility for future extensions.

(SL.1) A *field* is present only when the enclosing class is reachable.

(SL.2) A *method* is present only when the enclosing class is reachable.

(SK.1) A *constructor parameter* is present only when the enclosing class is reachable.

(SK.2) A *super constructor invocation parameter* is present only when the enclosing class is reachable.

(SK.3) A *field assignment* in a constructor is present only when the enclosing class is reachable.

(SM.1) A *method parameter* is present only when the enclosing method is reachable.

(ST.1) A *method invocation parameter* is present only when the enclosing term is reachable.

(ST.2) An *object creation parameter* is present only when the enclosing term is reachable.

In the remainder of this section, we highlight changes compared to the original FJ calculus for the annotation rules (L.1–T.5) in light gray and changes for the subtree rules (SL.1–ST.2) in darker gray.

––––––––––

[6]The names reference the according productions in CFJ's syntax in Figure 1. For example, K.1 is the first check that addresses the constructor.

$$\text{C} <: \text{C} \qquad \frac{\text{C} <: \text{D} \quad \text{D} <: \text{E}}{\text{C} <: \text{E}} \qquad \frac{\text{class C extends D } \{\,\dots\}}{\text{C} <: \text{D}}$$

<div align="center">Fig. 2.   CFJ subtyping.</div>

Field lookup $\boxed{\textit{fields}(\text{C}) = \overline{\text{C f}}}$

$$\textit{fields}(\text{Object}) = \bullet$$

$$\frac{CT(\text{C}) = \text{class C extends D } \{\,\overline{\text{C f}};\, \text{K } \overline{\text{M}}\,\} \quad \textit{fields}(\text{D}) = \overline{\text{D g}}}{\textit{fields}(\text{C}) = \overline{\text{D g}}, \overline{\text{C f}}}$$

Method lookup $\boxed{\textit{mtype}(\text{m}, \text{C}, \mathcal{A}) = \overline{\text{B x}} \rightarrow \text{B}}$

$$\frac{CT(\text{C}) = \text{class C extends D } \{\,\overline{\text{C f}};\, \text{K } \overline{\text{M}}\,\} \quad \text{M} = \text{B m}(\overline{\text{B x}}) \{ \text{ return t; } \} \quad \text{M} \in \overline{\text{M}} \quad \mathcal{A} \rightarrow AT(\text{M})}{\textit{mtype}(\text{m}, \text{C}, \mathcal{A}) = \overline{\text{B x}} \rightarrow \text{B}}$$

$$\frac{CT(\text{C}) = \text{class C extends D } \{\,\overline{\text{C f}};\, \text{K } \overline{\text{M}}\,\} \quad \text{M} = \text{B m}(\overline{\text{B x}}) \{ \text{ return t; } \} \quad \text{M} \in \overline{\text{M}} \quad \neg(\mathcal{A} \rightarrow AT(\text{M}))}{\textit{mtype}(\text{m}, \text{C}, \mathcal{A}) = \textit{mtype}(\text{m}, \text{D}, \mathcal{A} \wedge \neg AT(\text{M}))}$$

$$\frac{CT(\text{C}) = \text{class C extends D } \{\,\overline{\text{C f}};\, \text{K } \overline{\text{M}}\,\} \quad \text{m is not defined in } \overline{\text{M}}}{\textit{mtype}(\text{m}, \text{C}, \mathcal{A}) = \textit{mtype}(\text{m}, \text{D}, \mathcal{A})}$$

Overriding $\boxed{\textit{override}(\text{m}, \text{C}, \overline{\text{C x}} \rightarrow \text{C}_0, \mathcal{A})}$

$$\overline{\textit{override}(\text{m}, \text{Object}, \overline{\text{C x}} \rightarrow \text{C}_0, \mathcal{A})}$$

$$\frac{\begin{array}{c} CT(\text{C}) = \text{class C extends D } \{\,\overline{\text{D f}};\, \text{K } \overline{\text{M}}\,\} \\ \textit{override}(\text{m}, \text{D}, \overline{\text{C x}} \rightarrow \text{C}_0, \mathcal{A}) \quad \text{M} = \text{B}_0 \text{ m}(\overline{\text{B g}}) \{ \text{ return t; } \} \\ \text{M} \in \overline{\text{M}} \text{ implies } \overline{\text{C}} = \overline{\text{B}} \text{ and } \text{C}_0 = \text{B}_0 \text{ and } (\mathcal{A} \wedge AT(\text{M})) \rightarrow (AT(\overline{\text{C x}}) \leftrightarrow AT(\overline{\text{B g}})) \end{array}}{\textit{override}(\text{m}, \text{C}, \overline{\text{C x}} \rightarrow \text{C}_0, \mathcal{A})}$$

<div align="center">Fig. 3.   CFJ auxiliary functions.</div>

5.5. Typing

5.5.1. Subtyping. CFJ's subtyping relation $<:$, shown in Figure 2, is identical to FJ's. Though we could check the annotation rule (L.1) here, we decided to postpone this check to FJ's typing judgments instead (see T-CLASS).

5.5.2. Auxiliary Functions. As in FJ, we need some auxiliary definitions for the typing judgments shown in Figure 3. Although we try to perform most annotation checks in the typing judgments, there are cases in which already the auxiliary functions – that are used in FJ to recursively look up fields or methods across the inheritance hierarchy – need to evaluate annotations. We use $\mathcal{A}$ as a metavariable for annotations and use $\bullet$ to denote an empty sequence.

Field lookup. First, a *fields* determines all fields of a class C including fields inherited from superclasses. In CFJ, the function *fields* is identical to the one in FJ. Annotations on fields are checked later in the typing judgments.

Method lookup. Second, similar to the field lookup, the *mtype* finds methods with a given name m in a class C or its superclasses.[7] In contrast to fields, the method lookup needs to be adapted because of the possibility of method overriding (in contrast to overshadowing fields, which is not allowed in FJ [Igarashi et al. 2001]). Thus, it could be possible that a method m in class C is not always reachable for a given annotation $\mathcal{A}$, but another method m in a superclass of C is. Therefore, we cannot check annotations only in the typing judgments but have to adapt the auxiliary function *mtype* as shown in Figure 3.

In FJ, there are two possible cases, either the method is found in class C, then its signature is returned, or the method is not found, then the search proceeds to the superclass. In CFJ, we additionally have to distinguish whether found method is always reachable or not. Reachability is checked against a given annotation that is provided as a parameter $\mathcal{A}$ (i.e., $\mathcal{A} \rightarrow AT(\mathsf{M})$). In case it is not always reachable, the search is continued in the superclass for the remaining variants with a reduced annotation ($\mathcal{A} \wedge \neg AT(\mathsf{M})$). Note that auxiliary function *override*, as described below, checks that all these methods have compatible signatures; here, we check overridden methods only regarding reachability.

Overriding. Finally, the third auxiliary function *override* checks valid *method overriding* in FJ. In the presence of annotations, checking valid overriding is trickier than expected. We need to ensure that the return type and parameter types match in every variant in which two methods with the same name appear in the inheritance hierarchy of a class. This is complicated, because we allow developers to annotate both methods and their parameters.

Method overriding is the first and most important rule for which considerations regarding the desired backward compatibility – every CFJ product-line implementation stripped of its annotation should be a well-typed FJ program – have influenced design decisions. We describe our solution fulfilling this property first and discuss possible alternatives later.

Our function *override* works in the following way: for a given method m with annotation $\mathcal{A}$ and type $\overline{\mathsf{C}} \rightarrow \mathsf{C}_0$, we iterate over all superclasses until we reach Object. Whenever we find a method in a superclass with the same name, we perform the two checks. First, for backward compatibility, the return type and all parameter types must match independent of any annotation ($\mathsf{C}_0 = \mathsf{B}_0$ and $\overline{\mathsf{C}} = \overline{\mathsf{B}}$); this implies also that both methods have the same number of parameters. Second, for (M.2), in all variants in which both methods are present (i.e., for which both $\mathcal{A}$ and $AT(\mathsf{M})$ both evaluate to *true*) the annotations on parameters must be equivalent (formalized as $(\mathcal{A} \wedge AT(\mathsf{M})) \rightarrow (AT(\overline{\mathsf{C}\,\mathsf{f}}) \leftrightarrow AT(\overline{\mathsf{B}\,\mathsf{g}}))$). Taking both checks into account, we define the auxiliary function *override* as shown in Figure 3.

Due to our design decision for backward compatibility, our *override* function does not allow different signatures of a method in mutually exclusive features. For example, although the following code fragment generates only well-typed variants given that features *X* and *Y* are mutually exclusive, it is rejected by our *override* function.[8]

```
1  class D extends E { #ifdef X C f(C x) {...} #endif }
2  class C extends D { C f(#ifdef Y D y, #endif C x){...}}
```

Different typing judgments would be possible that drop backward compatibility in exchange for increased expressiveness. In such case, we would need to check valid overriding only when two methods can occur in the same variant. Since we pursue backward compatibility, we keep our simpler version of *override*. For developers this restricted expressiveness is not limiting since simple workarounds can be used; in the code example above, we could add a parameter D y to the first method declaration and annotate it such that it is never present in any variant (e.g., '*#if 0*').

---

[7]For technical reasons, we return the entire parameter list $\overline{\mathsf{B}\,\mathsf{x}}$ instead only their types, so that we can later (in rule T-INVK) reason about annotations on parameters.

[8]We leave out the constructor for conciseness in this example.

Term typing $\boxed{\Gamma \vdash t : C}$

$$\frac{x : C \; \textit{with} \; \mathcal{A}' \in \Gamma \quad \mathcal{A} \to \mathcal{A}'}{\mathcal{A};\Gamma \vdash x : C} \tag{T-VAR}$$

$$\frac{\mathcal{A};\Gamma \vdash t_0 : C_0 \quad \textit{fields}(C_0) = \overline{C\,f} \quad \mathcal{A} \to AT(C_i\,f_i)}{\mathcal{A};\Gamma \vdash t_0.f_i : C_i} \tag{T-FIELD}$$

$$\frac{\mathcal{A};\Gamma \vdash t_0 : C_0 \quad \textit{mtype}(m, C_0, \mathcal{A}) = \overline{D\,y} \to C \quad AT(\bar{t});\Gamma \vdash \bar{t} : \overline{C} \quad \overline{C} <: \overline{D}}{\mathcal{A} \to \big(AT(\bar{t}) \leftrightarrow AT(\overline{D\,y})\big) \quad AT(\bar{t}) \to \mathcal{A}}{\mathcal{A};\Gamma \vdash t_0.m(\bar{t}) : C} \tag{T-INVK}$$

$$\frac{\textit{fields}(C) = \overline{D\,f} \quad AT(\bar{t});\Gamma \vdash \bar{t} : \overline{C} \quad \overline{C} <: \overline{D}}{\mathcal{A} \to AT(C) \quad \mathcal{A} \to \big(AT(\bar{t}) \leftrightarrow AT(\overline{D\,f})\big) \quad AT(\bar{t}) \to \mathcal{A}}{\mathcal{A};\Gamma \vdash \text{new } C(\bar{t}) : C} \tag{T-NEW}$$

Method typing $\boxed{M \text{ OK in } C}$

$$\frac{\begin{array}{c} M = C_0\, m(\overline{C\,x})\,\{\,\text{return } t_0;\,\} \quad AT(M) = \mathcal{A} \quad \mathcal{A} \to AT(C_0) \quad AT(\overline{C\,x}) \to AT(\overline{C}) \\ CT(C) = \text{class } C \text{ extends } D\,\{\dots\} \quad \textit{override}(m, D, \overline{C} \to C_0, \mathcal{A}) \\ \Gamma = \bar{x} : \overline{C} \; \textit{with } AT(\overline{C\,x}), \text{this} : C \; \textit{with } AT(C) \quad \mathcal{A};\Gamma \vdash t_0 : E_0 \quad E_0 <: C_0 \\ AT(\overline{C\,x}) \to \mathcal{A} \end{array}}{M \text{ OK in } C} \tag{T-METHOD}$$

Class typing $\boxed{L \text{ OK}}$

$$\frac{\begin{array}{c} K = C(\overline{D\,g}, \overline{C\,f}')\,\{\,\text{super}(\overline{g}');\, \overline{\text{this.f=f}};\,\} \quad \overline{M} \text{ OK in } C \quad \textit{fields}(D) = \overline{D\,g}'' \\ \overline{C\,f} = \overline{C\,f}' \quad \overline{D\,g} = \overline{D\,g}'' \quad \overline{g} = \overline{g}' \\ AT(C) = \mathcal{A} \quad \mathcal{A} \to AT(D) \quad AT(\overline{C\,f}) \leftrightarrow AT(\overline{\text{this.f=f}}) \quad AT(\overline{C\,f}) \leftrightarrow AT(\overline{C\,f}') \\ \mathcal{A} \to \big(AT(\overline{D\,g}) \leftrightarrow AT(\overline{D\,g}'')\big) \quad AT(\overline{D\,g}) \leftrightarrow AT(\overline{g}') \quad AT(\overline{C\,f}) \to AT(\overline{C}) \\ AT(\overline{C\,f}) \to \mathcal{A} \quad AT(\overline{M}) \to \mathcal{A} \quad AT(\overline{D\,g}) \to \mathcal{A} \end{array}}{\text{class } C \text{ extends } D\,\{\,\overline{C\,f};\, K\,\overline{M}\,\} \text{ OK}} \tag{T-CLASS}$$

Product-line typing $\boxed{P \text{ OK}}$

$$\frac{\overline{L} \text{ OK} \quad ;\vdash t : C}{(\overline{L}, t) \text{ OK}} \tag{T-SPL}$$

Fig. 4.   CFJ typing.

5.5.3. Typing Judgments. For term typing and well-formedness rules, we revisit each typing judgment in FJ and adapt it for CFJ to incorporate annotations as shown in Figure 4. For brevity, we discuss only changes compared to FJ.[9]

For all typing judgments for terms, we need an environment that is extended by annotations. The environment Γ is a finite mapping from variables to pairs of a type and an annotation written

---

[9]Technically, it is possible to separate the CFJ type system into two parts: the original FJ type system and an extension for reachability checks on annotations. Such separation would follow our implementation and the idea behind backward compatibility. However, separated reachability checks replicate and adjust many mechanisms from FJ; it would almost double the length of the calculus. We present the shorter, integrated description of the CFJ type system instead.

$\bar{x} : \overline{C}$ *with* $\overline{\mathcal{A}}$. Additionally, the current annotation $\mathcal{A}$ is stored as environment. For the outermost term in a method, the current annotation is the annotation of a method (see T-METHOD); for inner terms, the current annotation may differ because parameters can be annotated individually (see T-INVK and T-NEW). The typing judgment for terms has the form $\mathcal{A}; \Gamma \vdash t : C$ and reads "in the environment Γ with annotation $\mathcal{A}$, term t has the type C".

When typing a variable (T-VAR), we need to ensure that the variable is reachable in all variants in which x is accessed. This means that we check reachability between the current annotation of the variable access $\mathcal{A}$ and the annotation $\mathcal{A}'$ of the parameter (or this) passed through the environment Γ from T-METHOD.

For typing field accesses (T-FIELD), we require that the target field declaration is reachable (T.2). Therefore, we check reachability between the current annotation $\mathcal{A}$ and the annotation of the target field ($AT(C_i \; f_i)$). The typing judgment for classes (see T-CLASS) ensures that the class corresponding to each field's type (C_i) is reachable (L.2).

For typing method invocations (T-INVK), we similarly check that a target method is present (T.3*i*) using the filtering of *mtype*. Parameters in method invocations can be annotated individually, so we need to check that the invocation parameters match the expected parameters of the method declaration in every variant (T.3*ii*). We use the same mechanism $\mathcal{A} \to \left(AT(\bar{t}) \leftrightarrow AT(\overline{D \; y}) \right)$ as for the *override* function (with the same implications for backward compatibility). Actually, in the presence of method overriding, there can be different target methods in different variants; *mtype* ensures that always at least one of these methods is available, and *override* (called in T-METHOD) ensures that overriding methods have compatible type signatures and compatible annotations on parameters. Furthermore, when typing a parameter, the annotation context is set to the annotation of this parameter ($AT(t_i); \Gamma \vdash t_i : C_i$). Finally, the subtree rule (ST.1) is checked: There must not be a variant in which the invocation is removed but not its parameter ($AT(\bar{t}) \to \mathcal{A}$).

Typing an object creation term (T-NEW) is similar to typing a method invocation. First, the target class must be present (T.4*i*), which is checked explicitly with $\mathcal{A} \to AT(C)$. Additionally for rule (T.4*ii*), we ensure that the provided parameters match the expected constructor parameters in every variant ($\mathcal{A} \to \left(AT(\bar{t}) \leftrightarrow AT(\overline{D \; f}) \right)$). Finally, the subtree rule (ST.2) is checked.

The typing judgment for method declarations (T-METHOD) has the form M OK in C and reads "method declaration M is well-formed, when it occurs in class C". We make several extensions shown in Figure 4: First, we check valid overriding in all variants (M.2) by passing the method's annotation to auxiliary function *override*. Second, we check that the class corresponding to the return type and all parameters of the method (C_0 and $\overline{C}$) are reachable (M.1, M.3).[10] Third, we provide the annotations of parameters in the type context to be checked in T-VAR later (T.1), and use the current annotation of the method $\mathcal{A}$ as annotation context. Finally, we check the subtree rule (SM.1).

The typing judgment for class declarations (T-CLASS) has the form L OK. At first, it appears very complex because it covers many annotation rules, but each rule by itself is simple. To distinguish the occurrences of $\bar{g}$ as constructor parameters, super invocation parameters, and fields of the superclass – which can all have different annotations – we distinguish $\bar{g}$, $\bar{g}'$ and $\bar{g}''$ but still assume that all $\bar{g}$'s are named the same ($\bar{g} = \bar{g}' = \bar{g}''$). The same for $\overline{C \; f}$ that is used both for fields and constructor parameters ($\overline{C \; f} = \overline{C \; f}'$). First, rule (L.1) checks that the superclass is always reachable ($\mathcal{A} \to AT(D)$); thus, from every reachable class, we can reach all its superclasses. Second, rule (K.1) specifies that the super-constructor call receives exactly those parameters from the constructor's parameter list that are defined as fields in the superclass in all variants ($AT(\overline{D \; g}) \leftrightarrow AT(\bar{g}')$ and $\mathcal{A} \to \left(AT(\overline{D \; g}) \leftrightarrow AT(\overline{D \; g}'') \right)$). Third, rule (K.2) specifies that the remaining constructor parameters match the field assignments and that those match the fields declared in the class ($AT(\overline{C \; f}) \leftrightarrow AT(\overline{this.f=f})$ and $AT(\overline{C \; f}) \leftrightarrow AT(\overline{C \; f}')$). Fourth, we check that the class corresponding to the type of each field in this class is reachable when the field is reachable ($AT(\overline{C \; f}) \to AT(\overline{C})$),

---

[10]Thüm proved that the check $\mathcal{A} \to AT(C_0)$ is actually redundant [Thüm 2010]. Still, we leave it for readability.

$$\boxed{remove(\overline{\mathsf{a}}, F), \text{ short } \langle\!\langle\overline{\mathsf{a}}\rangle\!\rangle}$$

$$remove(\overline{\mathsf{a}}, F) = \begin{cases} \mathsf{a}_1, remove(\mathsf{a}_2 \ldots \mathsf{a}_n, F) & \text{if } eval(AT(\mathsf{a}_1), F) \\ remove(\mathsf{a}_2 \ldots \mathsf{a}_n, F) & \text{else} \end{cases}$$

$$remove(\bullet, F) = \bullet$$

$$\boxed{variant(\overline{\mathsf{a}}, F), \text{ short } [\![\overline{\mathsf{a}}]\!]}$$

$$[\![\mathsf{x}]\!] = \mathsf{x} \tag{G.1}$$

$$[\![\mathsf{t.f}]\!] = [\![\mathsf{t}]\!].\mathsf{f} \tag{G.2}$$

$$[\![\mathsf{t.m}(\overline{\mathsf{t}})]\!] = [\![\mathsf{t}]\!].\mathsf{m}([\![\langle\!\langle\overline{\mathsf{t}}\rangle\!\rangle]\!]) \tag{G.3}$$

$$[\![\mathsf{new\ C}(\overline{\mathsf{t}})]\!] = \mathsf{new\ C}([\![\langle\!\langle\overline{\mathsf{t}}\rangle\!\rangle]\!]) \tag{G.4}$$

$$[\![\mathsf{C\ m}(\overline{\mathsf{C\ x}}) \{\mathsf{return\ t};\}]\!] = \mathsf{C\ m}(\langle\!\langle\overline{\mathsf{C\ x}}\rangle\!\rangle) \{\mathsf{return}\ [\![\mathsf{t}]\!];\} \tag{G.5}$$

$$[\![\mathsf{C}(\overline{\mathsf{C\ f}}) \{\mathsf{super}(\overline{\mathsf{f}}); \overline{\mathsf{this.f=f};}\}]\!] = \mathsf{C}(\langle\!\langle\overline{\mathsf{C\ f}}\rangle\!\rangle) \{\mathsf{super}(\langle\!\langle\overline{\mathsf{f}}\rangle\!\rangle); \langle\!\langle\overline{\mathsf{this.f=f};}\rangle\!\rangle\} \tag{G.6}$$

$$[\![\mathsf{class\ C\ extends\ D} \{\ \overline{\mathsf{C\ f}}; \mathsf{K}\ \overline{\mathsf{M}}\ \}]\!] = \mathsf{class\ C\ extends\ D} \{\ \langle\!\langle\overline{\mathsf{C\ f}}\rangle\!\rangle; [\![\mathsf{K}]\!]\ [\![\langle\!\langle\overline{\mathsf{M}}\rangle\!\rangle]\!]\ \} \tag{G.7}$$

$$[\![(\overline{\mathsf{L}}, \mathsf{t})]\!] = ([\![\langle\!\langle\overline{\mathsf{L}}\rangle\!\rangle]\!], [\![\mathsf{t}]\!]) \tag{G.8}$$

Fig. 5. CFJ variant generation with *remove* and *variant*.

which indirectly covers rules (L.2) and (K.3). Fifth, subtree rules for fields, methods and constructor parameters (SL.1–2, SK.1–3) are checked.

Finally, we are able to define when a software product line is well-typed (T-SPL): A software product line is well-typed if all of its classes are well-formed and the typing judgment returns a type for the start term t (provided an empty environment with an empty annotation, written as "$; \vdash \mathsf{t} : \mathsf{C}$").

5.6. Variant Generation

Although technically possible, we do not execute product lines written in CFJ directly. Thus, there are no evaluation rules for CFJ, and it is not possible or necessary to prove type soundness with the standard theorems progress and preservation [Wright and Felleisen 1994]. Instead, with a valid feature selection, we generate a tailored FJ programs by removing certain annotated code fragments. The resulting FJ program can be evaluated with FJ's evaluation rules (see [Igarashi et al. 2001]). For FJ, type soundness has already been proved [Igarashi et al. 2001]. Hence, we describe the variant generation mechanism and subsequently prove that generation preserves typing in Section 5.7.

To generate a program variant, we define a function *variant* that takes a CFJ product line *P* and a feature selection *F* as input and returns an FJ program. The function *variant* descends recursively through the code of the product line and applies a function *remove* to all code fragments that can be annotated. The function *remove* evaluates possible annotations (as described in Section 5.3): those code fragments, for which the annotation evaluates to *false* are removed, all other code fragments remain in the code.[11]

We define the generation rules (bottom-up) in Figure 5. For brevity, we write $variant(\mathsf{a}, F)$ as $[\![\mathsf{a}]\!]$ and $remove(\overline{\mathsf{a}}, F)$ as $\langle\!\langle\overline{\mathsf{a}}\rangle\!\rangle$ (we omit parameter F in the short form, because it is only propagated without modification).

---

[11] Since we describe annotations externally, we do not have to remove annotations explicitly during generation. Furthermore, in an implementation for a concrete language, *remove* must address the tokens used to separate list items (especially commas between parameters). In our tool CIDE, *remove* is implemented using transformations of the abstract syntax tree [Kästner et al. 2009].

5.7. Properties of CFJ

In Section 4, we discussed two desired properties: backward compatibility and generation preserves typing. With the presented type system and variant generation rules, we can now prove both properties for CFJ. Backward compatibility is straightforward to prove. Generation preserves typing is more complex, so we performed the proof with the proof assistant Coq; for brevity, here, we describe only the theorem and proof strategy.[12]

THEOREM 5.1 (BACKWARD COMPATIBILITY). *Every well-typed CFJ product line stripped of the feature model and all annotations (without removing any code fragments) is a well-typed FJ program.*

PROOF. CFJ has the same *syntax* as FJ. For stripping annotations, we assume that all annotations evaluate to *true* for all variants (i.e., $\forall F \, \forall a : eval(AT(a), F)$; called empty annotation). Now, we can prove that with empty annotations, the type systems of FJ and CFJ are equivalent: All reachability checks are always fulfilled; *mtype* in CFJ and FJ are equivalent considering that CFJ's *override* ensures the same method signature for all methods with the same name in a class hierarchy; and the remaining differences are straightforward to prove to be equivalent as well.   □

THEOREM 5.2 (GENERATION PRESERVES TYPING). *Every variant that is generated from a well-typed software product line P with a valid feature selection F is a well-typed FJ program.*

PROOF STRATEGY. We prove the theorem by induction on the structure of CFJ product lines, that is, induction over all possible CFJ class tables and all possible CFJ terms. Using induction, we recursively iterate over all elements of the CFJ class table (classes, methods, fields, parameter lists and terms) and the start term. For every CFJ element, if well-typed, we do an induction over the variant generation rules to determine all possibly generated FJ elements and prove that they are well-typed according to the FJ type system.[13] The proof that the generated element is a well-typed FJ element is specific for each different kind of element (e.g., class or method invocation). Generally speaking, we use the CFJ typing rules (including reachability conditions) and the variant generation mechanism to prove that all code elements needed to type a generated FJ element (e.g., referenced classes or methods) are part of the generated FJ program.

To illustrate the proof mechanism, consider the following example for the smallest element: an access to a variable. Variant generation for variables (G.1) is independent of the feature selection F and just returns this variable. Still, we have to prove that any generated FJ variable access is well-typed according to FJ's typing rules. FJ's typing rule T-VAR for variable access requires two conditions: (1) the provided environment Γ must not contain duplicates, and (2) the environment must contain the analyzed variable. For both conditions, we need to consider the FJ environment, which is formed by the enclosing generated method. Hence, we have to consider variant generation for methods, in which parameters can be removed (G.5). We can prove both conditions of FJ's T-VAR using induction on the environment:

(1) CFJ's type system forbids duplicates in parameter lists (cf. Sec. 5.2); thus, it forbids duplicates in the CFJ environment; variant generation can only remove entries (cf. Fig. 5); hence, all parameter lists generated from well-typed CFJ product lines are duplicate free.
(2) The generated variable always occurs in the FJ environment. This can be proved as follows: The variable access has been generated from a well-typed CFJ product line. In the well-typed CFJ product line, CFJ's T-VAR ensures that the variable occurs in the CFJ environment $\mathcal{A};\Gamma$ and that $\mathcal{A} \to \mathcal{A}'$, in which $\mathcal{A}'$ is the annotation of the corresponding CFJ method parameter. Additionally,

---

[12]Proof script available at `http://fosd.de/cfj/`; Thüm's Master's Thesis [Thüm 2010] contains a detailed description of the proof, its structure, and its strategies.

[13]In line with FJ, to support Java's mutually recursive types, we assume a fixed CFJ class table. For the same reason, we also assume that the feature selection is fixed so that variant generation produces a unique, fixed FJ class table. Still, since the proof covers arbitrary CFJ class tables and arbitrary feature selections, it holds for all CFJ product lines and all feature selections.

we know that $eval(\mathcal{A}, F)$ is *true*, because otherwise we would not have reached the current point (G.1) of variant generation (variant generation would have stopped in G.3, G.4, G.7, or G.8). Consequently, reachability $\mathcal{A} \rightarrow \mathcal{A}'$ implies that $eval(\mathcal{A}', F)$ is also *true*, so the parameter is not removed during variant generation; it is part of the FJ environment.

The proofs for other elements follows a similar pattern. They are often more complex, because more context information (other classes, methods, and fields) has to be considered. For example, due to overriding, a method invocation can point to different methods in different FJ variants; hence, the proof considers information from auxiliary function *overriding* in *T-Method*, which ensures that overriding methods always have compatible signatures. Nevertheless, the general proof pattern is the same: induction over well-typed CFJ elements and variant generation rules, proving that each generated FJ element is well-typed with information from the induction steps (and often induction over other elements). The entire proof is available as a script for Coq (see above).   □

A third interesting property of CFJ's type system is completeness: Given a software product line *P* and given that *all* valid feature selections *F* yield well-typed FJ programs according to Theorem 5.2, is *P* well-typed according to the CFJ typing judgments? Unfortunately, this property does not hold due to backward compatibility. It is possible to find an ill-typed CFJ product line, of which only well-typed variants are generated; for an example consider the discussion about different parameters in Section 5.5.2. That is, due to our decision for backward compatibility, CFJ is stricter than actually necessary. Nevertheless, as discussed before, we decided to enforce these restrictions for the benefit of tool developers. Still, with tests and our case studies (see Section 8), we confirm that CFJ is not too strict for practical applications.

6. ALTERNATIVE FEATURES

Our tool CIDE has its roots in decomposing legacy applications. In the formalization of CFJ, these roots are visible. It is possible to make code fragments optional and to express annotations like *either FeatureA or FeatureB must be selected*. However, in CFJ it is difficult to have two alternative (mutually exclusive) implementations of the same class or method, similar to the persistent vs. in-memory storage example in Section 3. Since we want CFJ to be backward compatible, we cannot simply allow multiple classes or members with the same name (and signature) because this is not supported by FJ (and Java). Nevertheless, alternative features are used in software product lines, when a common implementation expects to reach exactly one (of multiple alternative) implementations of a class or method. Thus, for product-line development in general, we need to provide a way to implement and type check alternative features.

Alternative features may influence the implementation in different locations:

(1) *Alternative Classes.* Depending on the feature selection, there may be entirely alternative implementations of a class. Different implementations may contain different methods, common methods, or different implementations of the same method. They may even have nothing in common except the class's name, as long as both classes are annotated to be mutually exclusive.
(2) *Alternative Members.* There can be different methods with the same name, but different bodies, parameters, and return types. Thus, depending on the feature selection, a method may be implemented differently, even with different signatures.
(3) *Alternative Terms.* There can be different implementations of a method body, or alternative terms passed as parameters of a method invocation depending on the feature selection. Thus, it is also necessary to discuss alternative implementations of a term, not only of classes or methods.

6.1. Reduction to Alternative Terms

There are different strategies how to deal with alternative features (in CFJ and in practice). One useful strategy is to reduce alternative implementations to alternatives at the term level (respectively at statement level in Java). For CFJ, the reduction proceeds in two steps – merging classes and

```
1  class Database { ... }
2  #ifdef PERSISTENT
3  class Storage {
4      boolean save() { /* impl. A */ }
5      boolean clear() { /* impl. B */ }
6      boolean set(Object key, Object data, Lock lock) {
7          return /* impl. C */; }
8  }
9  #endif
10 #ifdef INMEMORY
11 class Storage {
12     boolean clear() { /* impl. B */ }
13     boolean set(Object key, Object data) {
14         return /* impl. D */; }
15 }
16 #endif
```

$$\Downarrow$$

```
1  class Database { ... }
2  #ifdef PERSISTENT ∨ INMEMORY
3  class Storage {
4  #ifdef PERSISTENT
5      boolean save() { /* impl. A */ }
6  #endif
7      boolean clear() { /* impl. B */ }
8      boolean set(Object key, Object data
9          #ifdef PERSISTENT, Lock lock#endif) {
10     return #ifdef PERSISTENT/* impl. C */#endif
11         #ifdef INMEMORY/* impl. D */#endif;
12     }
13 }
14 #endif
```

Fig. 6.   Reducing alternative classes and alternative methods to alternative terms

merging members – and can be done by the developer or be automated by a tool (limitations of
these steps are written in square brackets and discussed subsequently):

— *Merging classes.* When there are two or more classes with the same name [and same superclass,
   see below] but different implementations and annotations, they can be all merged into one class.
   The new class is annotated with a disjunction of all individual annotations ($\mathcal{A}_1 \vee \mathcal{A}_2 \vee \ldots \vee \mathcal{A}_n$),
   so that it is present in a variant if any of the original classes would be present. All members from
   the original classes are moved into the merged class and keep their annotations (the subtree rules
   (SL.1) and (SL.2) are automatically fulfilled). This step reduces alternative classes to alternative
   methods in a single merged class.
— *Merging members.* When there are two or more methods with the same name [and return type,
   see below] in a single class declaration, they can be merged to a single method annotated with
   a disjunction of all previous annotations. Parameters also are merged and annotated with a dis-
   junction of all previous annotations of each parameter. If their bodies are not the same, we need
   a way to represent alternative terms inside this method. Analogously, multiple fields with the
   same name [and type, see below] can be merged. This way, we reduce alternative methods to
   alternative terms.

In Figure 6, we show this reduction for an extended example of the persistent vs. in-memory storage
from Section 3. We reduce two alternative implementations of the class *Database* to a single class
and two alternative implementations of method *set* with different parameters to a single method
with alternative terms.

   The reduction to alternative terms is limited regarding superclasses, return types, and field types.
That is, if two alternative classes with the same name do not have the same superclass, if two meth-

```
1  class Storage₁ extends Object {
2    #ifdef 𝒜₁ boolean set() {return /*impl.1 */;} #endif
3  }
4  class Storage₂ extends Storage₁ {
5    #ifdef 𝒜₂ boolean set() {return /*impl.2 */;} #endif
6  }
7  //...
8  class Storage extends Storageₙ₋₁ {
9    #ifdef 𝒜ₙ boolean set() {return /*impl.n */;} #endif
10 }
```

Fig. 7.   Implementing alternative return terms with method overriding.

ods with the same name do not have the same return type, or if two fields with the same name do not have the same type, they cannot always be merged. We can either accept this limitation and disallow the three problematic cases, or we can search for mechanisms that support alternative implementations beyond alternative terms. To retain backward compatibility and since such cases are rare in practice (usually alternative implementations of a class still provide a common interface), we accept the limitation and suggest workarounds instead of new language features such as multiple inheritance. A simple workaround, which works for all three problems, is to rename classes, methods, or fields with fresh names. By renaming the target declarations, variability is again propagated to alternative terms where depending on the feature selection either of the now distinguishable methods is invoked, either of the fields is accessed, or either classes is instantiated. For CFJ and our implementation for Java, we prefer to accept this limitation – enforcing constant superclasses, return types, and field types in all alternative implementations of a class method or field – and use the renaming workaround (which can even be automated) for all other cases, instead of complicating the type system. Nevertheless, other solutions without these limitations but with more complex typing judgments are possible, see Section 9.

6.2. Handling Alternative Terms

So far, we reduced the problem of alternative implementations to alternative terms (in CFJ) or alternative statements (in Java and many other languages). Now, we have to make sure that parser and type checker understand alternative terms/statements and check them accordingly.

In CFJ, the situation is especially problematic, since every method must contain exactly one return statement (i.e., a single term). We must make sure that in every variant exactly one (not none, not multiple) of these terms remains. For CFJ, we discuss three solutions; although the first two have significant drawbacks, we briefly summarize all three here:

— *Method overriding.* Without changes to the CFJ calculus, we found only one way to implement alternative terms. The basic idea is to create an artificial superclass for each alternative term and use method overriding to provide different terms in different classes as illustrated in Figure 7. In such implementation, the target method has a different annotation in each subclass, and in a generated variant only one of these methods remains (auxiliary function *mtype* ensures that always at least one of these methods is present). Although this approach can be used without modification of CFJ and is backward compatible to FJ, it has the drawback of significantly obfuscating the source code with boilerplate code.
— *New language constructs.* A whole group of solutions for alternative terms becomes available once we drop backward compatibility and decide to change the syntax or typing judgments of CFJ. For example, we could simply allow two methods with the same name or a method with two return statements and adjust the syntax and typing judgments to ensure that at most one of them remains in a generated variant. Another solution is to introduce new language constructs which allow refinements of classes or methods. That is, we could integrate language mechanisms such as mixins [Bracha and Cook 1990; Flatt et al. 1998], class refinements [Batory et al. 2004; Apel et al. 2008], aspects [Kiczales et al. 1997], classboxes [Bergel et al. 2005], traits [Ducasse

```
 1  class Database { ... }
 2  class Storage {
 3      boolean set(Object key, Object data, Lock lock) {
 4          boolean result;
 5  #ifdef PERSISTENT
 6          result = /* implementation A */
 7  #endif
 8  #ifdef INMEMORY
 9          result = /* implementation B */
10  #endif
11          return result;
12      }
13  }
14  }
```

Fig. 8.   Rewritten example of alternative return statements.

et al. 2006], hyperslices [Tarr et al. 1999], and others. These approaches are interesting when designing a completely new language – in fact, in a different line of research, we designed a product-line–aware type system for class refinements [Apel et al. 2010] – however in this work, we prefer a backward compatible solution that is easier to adopt in practice.

— *Metaexpressions.* Czarnecki and Antkiewicz [2005] suggested metaexpressions as a mechanism to support alternative values in a software product line of UML models. In their setting, they did not have the opportunity to change the syntax of UML but sought for another way to express alternatives. Metaexpressions are annotations, stored separately, which specify one or more alternative values for a language construct like the name of an UML association. This means, instead of changing the syntax, alternatives are specified externally by a tool. Then, the generation mechanism does not only remove code fragments for which annotations evaluate to *false*, but it can also replaces those elements with a metaexpression by their according value. The key difference to additional language constructs is that alternatives are specified externally on a tool level, but still checked by the type system (like the annotation table).

For full Java and many other languages, there are simpler solutions because these languages support multiple statements inside a method, so the desired backward compatibility does not impose so many restrictions. Having two statements in a method with alternative annotations is still backward compatible. In Java, only return statements are problematic, because of Java's unreachable code detection (code after a return statement results in a compiler error). Still, simple workarounds are possible, for example, we can rewrite the persistent vs. in-memory example from Section 3 as shown in Figure 8. In our experience with Java, all alternative features can be reduced to alternative statements and implemented without language extensions using such rewrites.

Despite the practical solution in full Java, we take a closer and formal look at metaexpressions for CFJ, to explore a solution for Featherweight Java and for potential other languages in which it is not possible to use statements as described above.

6.3. Metaexpressions

We describe metaexpressions with an external metaexpression table *MXT*, similar to the annotation table *AT*. Like annotations, we could introduce metaexpressions in the language's syntax, but we prefer to leave CFJ's syntax unmodified for backward compatibility.

The metaexpression table provides a list of alternatives for each term (again able to distinguish between multiple terms with the same name; e.g., identified by location); if a term does not have alternatives, MXT returns the empty list. We use the following notation to access alternatives and a specific alternative entry:

$$MXT(\mathrm{t}) = \mathrm{t}_1, \mathrm{t}_2, ..., \mathrm{t}_n$$

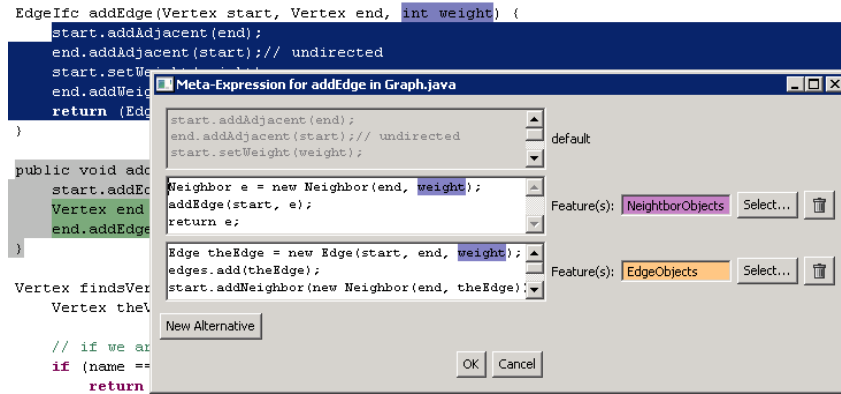$$MXT(\mathrm{t}, i) = \mathrm{t}_i$$

Fig. 9.   Editor for metaexpressions: In a code fragment of an product line of graph data structures, we select the entire body of a method and invoke the metaexpression editor from the context menu. In this editor, we specify alternative implementations to the selected code fragment.

Each of the alternatives can be annotated the usual way. Additionally, for inner terms of an alternative, the metaexpression table may provide alternatives again; as sanity condition, we only require that there are not cycles in the metaexpression table. In Figure 9, we illustrate how metaexpressions may be represented in a source-code editor for full Java: The user selects a code fragment and can provide alternative code fragments including annotations for these alternatives. Note that this solution is entirely backward compatible; instead of introducing a language construct, we provide alternatives externally with tool support.

During variant generation, for each term, we look up whether the term has alternatives; if it has and the annotation of an alternative evaluates to *true*, we replace the term by this alternative. Alternatives are ordered; in case annotations of multiple alternatives evaluate to *true*, the first is chosen. Regarding term typing, we need to ensure that all alternatives have the same type as (or a subtype of) the original term's type; that is, we ensure that alternatives are always substitutable for the original term.

6.3.1. Typing. To describe variant generation and typing formally, we have to make a number of changes to the calculus. During term typing, every time we derive the type of a term $(\mathcal{A}, \Gamma \vdash t : C)$ we have to consider potential alternatives. We therefore introduce a metaexpression-aware typing judgment written as $\mathcal{A}, \Gamma \vdash^{mx} t : C$; it reads "in the environment Γ with the current annotation $\mathcal{A}$, term t and all its alternatives have a type that is a subtype of C". Optionally, we may also report a warning or an error, if annotations of two or more alternative terms evaluate to *true* in the same variant, although the variant-generation mechanism already ensures that exactly one term is present in every variant.

The new typing judgment checks the original term t as before, but additionally determines the type of all alternatives as shown in Figure 10 (first judgment). The judgment returns the type that is the most-specific supertype of all alternatives (smallest upper bound, determined with function *sub* with standard semantics). Finally, we have to adjust the typing judgments T-FIELD, T-INVK, T-NEW, T-METHOD, and T-SPL to use $\vdash^{mx}$ instead of $\vdash$, as shown in Figure 10.

The type system with metaexpressions is stricter than the original type system of CFJ, because we always type check the original statement, but additionally also check alternatives. Alternatives can make the type of a term less specific (in the worst case, when all alternatives have unrelated types, the term has the least-specific type Object).

The choice that the term of a type with alternatives is the most specific supertype of all alternatives' types was a deliberate design decision. It provides the same expressiveness as the implementation pattern in Figures 7 and 8. Beyond that, we could allow that a term can have alternative types

$$\frac{\mathcal{A};\Gamma \vdash t : C_0 \quad \mathcal{A};\Gamma \vdash MXT(t,1) : C_1 \quad \ldots \quad \mathcal{A};\Gamma \vdash MXT(t,n) : C_n}{\mathcal{A};\Gamma \vdash^{mx} t : sub(C_0,C_1,\ldots,C_n)}$$

$$\frac{C_1 <: C_0 \quad C_2 <: C_0 \quad \ldots \quad C_n <: C_0}{sub(C_0,\overline{C}) = C_0}$$

$$\frac{CT(C_0) = \text{class } C_0 \text{ extends } D \quad \neg\big(C_1 <: C_0 \quad C_2 <: C_0 \quad \ldots \quad C_n <: C_0\big)}{sub(C_0,\overline{C}) = sub(D,\overline{C})}$$

$$\frac{\mathcal{A};\Gamma \vdash^{mx} t_0 : C_0 \quad fields(C_0) = \overline{C}\,f \quad \mathcal{A} \to AT(C_i\,f_i)}{\mathcal{A};\Gamma \vdash t_0.f_i : C_i} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\mathcal{A};\Gamma \vdash^{mx} t_0 : C_0 \quad mtype(m,C_0,\mathcal{A}) = \overline{D}\,\overline{y} \to C \quad AT(\overline{t});\Gamma \vdash^{mx} \overline{t} : \overline{C} \quad \overline{C} <: \overline{D} \\ \mathcal{A} \to \big(AT(\overline{t}) \leftrightarrow AT(\overline{D}\,\overline{y})\big) \quad AT(\overline{t}) \to \mathcal{A}\end{array}}{\mathcal{A};\Gamma \vdash t_0.m(\overline{t}) : C} \qquad \text{(T-INVK)}$$

$$\frac{\begin{array}{c}fields(C) = \overline{D}\,f \quad AT(\overline{t});\Gamma \vdash^{mx} \overline{t} : \overline{C} \quad \overline{C} <: \overline{D} \\ \mathcal{A} \to AT(C) \quad \mathcal{A} \to \big(AT(\overline{t}) \leftrightarrow AT(\overline{D}\,f)\big) \quad AT(\overline{t}) \to \mathcal{A}\end{array}}{\mathcal{A};\Gamma \vdash \text{new } C(\overline{t}) : C} \qquad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}M = C_0\,m(\overline{C}\,x)\,\{\text{ return } t_0; \} \quad AT(M) = \mathcal{A} \quad \mathcal{A} \to AT(C_0) \quad AT(\overline{C}\,x) \to AT(\overline{C}) \\ CT(C) = \text{class } C \text{ extends } D\,\{\ldots\} \quad override(m,D,\overline{C} \to C_0,\mathcal{A}) \\ \Gamma = \overline{x} : \overline{C} \; with \; AT(\overline{C}\,x), \text{this} : C \; with \; AT(C) \quad \mathcal{A};\Gamma \vdash^{mx} t_0 : E_0 \quad E_0 <: C_0 \\ AT(\overline{C}\,x) \to \mathcal{A}\end{array}}{M \text{ OK in } C} \qquad \text{(T-METHOD)}$$

$$\frac{\overline{L} \text{ OK} \qquad ;\vdash^{mx} t : C}{(\overline{L}, t) \text{ OK}} \qquad \text{(T-SPL)}$$

Fig. 10.   CFJ typing with metaexpressions (changes only).

depending on the feature selection (the metaexpression-aware typing judgment would return a list of types). However, alternative types depending on the feature selection would make type checking more complex and slower, and could lead to a combinatorial explosion of different alternatives. Alternative types can propagate through the entire type derivation process, for example x.f can have alternative types when x has alternative types. Furthermore, when a term can have one of many types, type errors become difficult to understand for users. We have explored this path and its consequences on complexity in an different product-line type system FFJ$_{PL}$ [Apel et al. 2010] as we discuss in Section 9. Here, we settle with the slightly less expressive, but simpler solution, which, in our opinion, is easier to handle for developers.

6.3.2. Variant Generation. The variant generation mechanism is changed similarly to the type system. In addition to the function $variant(a,F)$ (short $[\![a]\!]$), we need a metaexpression-aware function for terms $variant^{mx}(t,F)$ (short $[\![t]\!]^{mx}$). The function $variant^{mx}$ replaces the original term by the first alternative of which the annotation evaluates to *true*; if there is no alternative or all annotations of alternatives evaluate to *false*, the original term remains. Again, as for term typing, we need to adjust a number of variant generation rules (G.2, G.3, G.4, G.5, and G.8) to use the new variant function. We show the new function $variant^{mx}$ and the changed generation rules in Figure 11.

$$[\![t]\!]^{mx} = \begin{cases} [\![MXT(\text{t},1)]\!] & \text{if } eval(AT(MXT(\text{t},1)),F) \\ [\![MXT(\text{t},2)]\!] & \text{else if } eval(AT(MXT(\text{t},2)),F) \\ \dots & \\ [\![MXT(\text{t},n)]\!] & \text{else if } eval(AT(MXT(\text{t},n)),F) \\ [\![t]\!] & \text{else} \end{cases}$$

$$[\![\text{t.f}]\!] = [\![t]\!]^{mx}.\text{f}$$

$$[\![\text{t.m}(\bar{\text{t}})]\!] = [\![t]\!]^{mx}.\text{m}(\langle\!\langle[\![\bar{\text{t}}]\!]^{mx}\rangle\!\rangle)$$

$$[\![\text{new C}(\bar{\text{t}})]\!] = \text{new C}(\langle\!\langle[\![\bar{\text{t}}]\!]^{mx}\rangle\!\rangle)$$

$$[\![\text{C m}(\overline{\text{C x}}) \text{ \{return t;\}}]\!] = \text{C m}(\langle\!\langle\overline{\text{C x}}\rangle\!\rangle) \text{ \{return } [\![t]\!]^{mx};\}$$

$$[\![(\overline{\text{L}},\text{t})]\!] = ([\![\langle\!\langle\overline{\text{L}}\rangle\!\rangle]\!], [\![t]\!]^{mx})$$

Fig. 11.   CFJ variant generation with metaexpressions (changes only).

6.3.3. Properties. Both properties backward compatibility and generation preserves typing still hold. Backward compatibility is obvious, because we did not change the syntax and because the type system behaves just as the original CFJ type system when the metaexpression table is empty. Also generation preserves typing holds; the intuition is that $\vdash^{mx}$ checks all alternatives that can be generated by *variant^{mx}*. Again, we formalized our extensions and proved the theorem generation preserves typing with the proof assistant Coq.[14]

6.4. Summary

There are many different possibilities how alternative features can be implemented and type checked in a product line. Merging alternative classes and methods is not necessary but reduces the difficulty of finding mechanisms for implementation and type checking to alternative terms. In full Java, in which a method can contain a list of statements, we can now use alternative statements without further extensions. In FJ, we are more constrained because each method contains only a single return statement. While already without modifications of CFJ method overriding can be used as a 'hack', we prefer a dedicated extension of the type system. There are many novel language constructs we could introduce – mixins, aspects, traits, and many more – but these require significant changes to syntax and type system and are not backward compatible to FJ (and Java). To achieve backward compatibility to keep existing tool support, we introduce and type check metaexpressions as originally suggested for UML models by Czarnecki and Antkiewicz [2005]. Metaexpressions are added using an external metaexpression table and are backward compatible to Java in the sense that a metaexpression can always be added and type checked on top of an existing program. The main challenge remains to find an appropriate visualization for the editor that can convey the metaexpression concept to developers, potentially even including nested metaexpressions. However, a proper visualization is outside the scope of this paper.

7. BEYOND FEATHERWEIGHT JAVA (IMPLEMENTATION)

Our formalization is based on Featherweight Java because it allows proving the feasibility of a product-line–aware type system in a confined setting. Nevertheless, for a practical application, a product-line–aware type system should be provided for full Java or other languages. Our experience with CFJ guides the way for a more general implementation in our product-line tool CIDE.

CIDE is an Eclipse plug-in for product-line development. After specifying features in a feature model, a developer can assign annotations to code fragments. CIDE follows a model of disciplined annotations, in which annotations have to align with the underlying structure as outlined in Sec-

---

[14]Proof script available at `http://fosd.de/cfj/`

tion 5.2. CIDE represents annotations visually with background colors and provides various forms of additional tool support [Kästner et al. 2008], which are beyond the scope of this paper.

The formalization shows that backward compatibility is possible; we only have to add additional reachability checks between pairs (or triples or quadruples) of code fragments and their annotations. At a practical level, to achieve language independence (or at least extensibility toward new languages), we implemented a framework for product-line–aware type checking in CIDE that provides a general mechanism to iterate over a project, check reachability conditions, and report errors. CIDE displays detected errors like standard Java errors directly at their location (e.g., underlining a method invocation), and provides suggestions for fixing them. Our framework can be extended with plug-ins for specific languages. Each plug-in is responsible for determining which reachability conditions to check in a given language; for example, it looks up method invocations and corresponding method declarations. It is even possible check reachability conditions between elements of different languages (inter-language typing).

Currently, we provide the following type-checking plug-ins for CIDE:

— Featherweight Java. We implemented the CFJ type system in CIDE, including a metaexpression extension for alternative features (see Sec. 6.2). Specifically, Rosenthal [2009] implemented the entire type system natively without reusing an existing implementation.
— Java. For Java, we implemented all checks from Featherweight Java and several additional checks regarding local variables, interfaces, generics, imports, abstract classes, abstract methods, and others. This type system was implemented on top of Eclipse's type checks for Java, that is, we reused existing lookup mechanisms and added only reachability checks on top. To be precise, we could not reuse all lookup mechanisms, but had to slightly adapt those that are equivalent to *mtype* and *override* in Section 5.5.2. Although our implementation is probably not complete (a guarantee is difficult to provide for full Java), we believe that we have covered the most important causes of type errors and that our implementation is useful in practice.
The product-line–aware extension for Java is built on top of Eclipse's standard Java compiler. Thanks to backward compatibility, the existing syntax- and type checking mechanisms, the internal Java model, and the background compilation process of Eclipse remain untouched. Therefore, Eclipse provides tool support such as syntax highlighting, code completion, and code navigation; and Eclipse already detects all type errors of standard Java, we only add reachability checks on top.
— Bali. Bali is a grammar specification language in the AHEAD tool suite [Batory et al. 2004], for which we added reachability checks between references to and declarations of productions and tokens. In this language, looking up pairs is straightforward with a simple name table. Still, the entire mechanism to check reachability in the context of a feature models is reused and shared with the other languages.
— OSGi Manifest + Java. As a demonstration of inter-language typing, we implemented a plug-in that looks up package references between a manifest file of an OSGi bundle [OSGi Alliance 2009] and the bundle's implementation with Java. It again checks that the implementation is reachable from the according declaration in all variants, so that, in this case, no variant of an OSGi bundle can declare to export a package that it does not contain. So far, we implemented only checks for the *Export-Package* declaration as a proof of concept, but this can be extended easily to other checks between an OSGi manifest and Java or inter-language checks between other languages.

Together with an industrial partner, we are currently also implementing a product-line–aware type system for C that is largely backward compatible to the C preprocessor. This type system is developed outside CIDE, but follows the same mechanisms.

Finally, the mechanism to actually reason about feature models and annotations (to determine whether $AT(\mathsf{a}) \rightarrow AT(\mathsf{b})$ holds for all valid variants) also is abstracted behind an interface so that different reasoning mechanisms can be plugged in. Currently, we have implemented two mechanisms: a very simple one based on set relations (which however supports only very simple feature

models that can only express dependencies in the form of parent-child relationships in a tree, but no alternatives) and one for full feature models, originally developed for FeatureIDE [Leich et al. 2005; Kästner et al. 2009]. In the latter, which we use by default, reasoning is performed by transforming the feature model and reachability conditions into Boolean satisfiability problems as described by Batory [2005]; we subsequently solve the problem with the off-the-shelf SAT solver *SAT4J*.

To summarize, the formalization of CFJ is tailored to Featherweight Java, but the underlying mechanisms are general and can be transferred to other languages. Currently, the additional reachability checks for every language (and combination of languages in case of inter-language typing) are be provided manually using plug-ins. Whether these plug-ins can be generated automatically (e.g., from attribute grammars) is an open research question. Regarding inter-language typing, further research is needed to find the right abstractions (e.g., [Apel and Hutchins 2010]) or a suitable polylingual type system (e.g., [Grechanik et al. 2004]). From a tool perspective, recent advances in inter-language refactorings in Eclipse can be used as possible starting point [Fuhrer et al. 2007].

8. EVALUATION

In the previous sections, we have designed, formalized, and implemented a product-line–aware type system. To demonstrate its practicality, we performed a series of case studies to evaluate whether we can actually find type errors in existing product lines. Specifically, we want to answer the following questions:

— What are typical shapes of annotations?
— Does type checking detect relevant errors in software product lines?
— What performance can we expect from type checking a software product line (especially since Boolean satisfiability problems are involved)?

We applied our type checking approach to four case studies. As case studies, we selected Java programs that implement variability using some form of preprocessor. Since Java does not have a build in preprocessor, there are not as many projects as in C or C++, but, interestingly, providing variability is essential in the domain of software for mobile phones, so we found some open source projects that use the Java ME preprocessor *Antenna*.[15] We selected the following software product lines (see also Table I):

(1) *MobileMedia.* MobileMedia is a Java ME application to manipulate photo, music, and video files on mobile devices. It has been developed at Lancaster University as a product line and has been used in several studies on comparing conditional compilation with aspect-oriented mechanisms [Figueiredo et al. 2008; Conejero et al. 2009]. The product line has several optional features implemented with *#ifdef* directives, such as support for photos, music, video, SMS transfer, or favorites. We selected this product line because the code is peer reviewed [Figueiredo et al. 2008] and because the development is well documented in several incremental releases (each added one or more features), which allowed us to analyze simple as well as more complex versions. Specifically, we look at two releases: Release 6 with nine features and the latest Release 8 with 14 features (cf. [Figueiredo et al. 2008]).[16]
(2) *Mobile RSS Reader.* Mobile RSS Reader is an open source project to implement a portable RSS reader for mobile phones on the Java ME platform.[17] Variability is crucial to support different

---

[15]Antenna (`http://antenna.sourceforge.net/`) uses *#ifdef* directives very similar to the C preprocessor; however, Antenna's directives are written in comments. When running Antenna with a given feature set, it comments out all code of unselected features. The preprocessor is integrated in Java ME extensions of IDEs like Eclipse and NetBeans, in the latter even with additional syntax highlighting.

[16]The source code is available online at `http://mobilemedia.cvs.sf.net/viewvc/mobilemedia/`, of both releases, we used the code revision from July 9th, 2009.

[17]`http://code.google.com/p/mobile-rss-reader/`; Mobile RSS Reader under continuous development, we used revision 1596 (May 21st, 2009) available at `http://mobile-rss-reader.googlecode.com/svn/!svn/bc/1596/trunk/`.

Table I. Size and features of our case studies

Software product line	LOC	#FEA	#ANN	Features
MobileMedia (Rel. 6)	4 600	9	88	PHOTO, MUSIC, SMS, SORTING, COPYMEDIA, FA-VORITES, 128x149, 132x176, and 176x205
MobileMedia (Rel. 8)	5 700	14	164	PHOTO, MUSIC, VIDEO, SMS, SORTING, COPY-MEDIA, FAVORITES, PRIVACY, CAPTUREPHOTO, CAPTUREVIDEO, PLAYVIDEO, 128x149, 132x176, and 176x205
Mobile RSS Reader	20 000	14	1 050	MIDP10, MIDP20, JSR75, JSR238, CLDC11, SMALLMEM, ITUNES, LOGGING, TEST, TESTUI, 4×COMPATIBILITY
Lampiro	45 000	11	108	MOTOROLA, TLS, COMPRESSION, BXMPP, SCREENSAVER, UI, GLIDER, BLUDENO, TIMING, SENDDEBUG, and PLAINSOCKET
Berkeley DB	70 000	42	1 825	TRANSACTIONS, STATISTICS, DELETEDB-OPERATION, ENVIRONMENTLOCK, FILEHAN-DLECACHE, ... (see [Kästner et al. 2007] for a comprehensive list)

LOC: lines of code (approximated); #FEA: number of features; #ANN: number of annotated code fragments

devices, therefore typical features refer to Java ME libraries: MIDP 1.0, MIDP 2.0, CLDC 1.1, JSR 75 (file system), and JSR 238 (internationalization). Additional features include support for devices with small memory capacity, logging and testing features, and several compatibility features for different RSS formats.

(3) *Lampiro.* Lampiro is an instant-messaging Java ME client for the XMPP protocol developed by *Bluendo s.r.l.*, released as open source.[18] Several features, such as COMPRESSION, ENCRYP-TION (TLS), PROFILING and DEBUGGING, or SCREENSAVER, are implemented using *#ifdef* directives.

(4) *Berkeley DB.* Finally, Oracle's Berkeley DB is an open-source database engine written in Java, which we decomposed into features in prior work [Kästner et al. 2007; Kästner et al. 2008].[19] Berkeley DB is different from the case studies above in two ways. First, it was not originally developed as a product line, but we later refactored it into features, such as TRANSACTIONS, STATISTICS, ENVIRONMENTLOCK, or DELETEDBOPERATION. Second, we annotated the code base with CIDE after having implemented an initial version of our type system. This gives a different perspective on our type system regarding the development of a new product line by decomposing a legacy application.

8.1. Shape of annotations

In all case studies, annotations are used often at a fine granularity. While also entire classes and methods are annotated, most annotations are on statement level. In Mobile RSS Reader and Berkeley DB, even parameters in method declarations and method invocations were annotated. This fine granularity is where annotations play to their strength, compared to contemporary modularization techniques such as components or aspects [Kästner et al. 2008], but also where it is difficult to enforce reachability conditions manually due to their high number.

Most annotations were simple and consisted only of a single feature *(#ifdef X)* or a negated feature *(#ifndef X)*; however, nesting was quite common (up to level 4 in Mobile RSS Reader). Beyond single features and nesting, only MobileMedia used some pattern like $A \land B$ or $A \lor B$ (the most complex annotation we found was '(MUSIC $\land$ PHOTO) $\lor$ (MUSIC $\land$ VIDEO) $\lor$ (VIDEO $\land$ PHOTO)' in MobileMedia Release 8). Usually it is quite easy to reason about reachability manually and thus

---

[18] http://lampiro.bluendo.com/; Lampiro is still under development, we used version 9.6.0 (June 19th, 2009) available at http://lampiro.googlecode.com/svn/!svn/bc/30/trunk/.

[19] Specifically, we used Berkeley DB version 2.1.30 available at http://www.oracle.com/technology/software/products/berkeley-db/je/index.html.

interpret the errors reported by the type system. Nevertheless, automatically checking reachability constraints in a type system is helpful due to the sheer number of reachability constraints (up to 72 534 in Lampiro, cf. Tab. II).

In all software product lines that were developed with *#ifdef* directives originally, we found alternative features or alternative implementations depending on whether a feature is selected. Alternatives generally occurred on the level of statements or for setting initial values of constants. In Mobile RSS Reader, also alternative superclasses were used, so that a class inherits from different classes depending on whether feature TESTUI is selected. To avoid complexity, we forbid alternative superclasses (see discussion in Sec. 6.1) and rewrote the corresponding implementation. In general, we found 3 alternative code fragments in MobileMedia Release 6, 8 in MobileMedia Release 8, 70 in Mobile RSS Reader, and 10 in Lampiro.

8.2. Detecting Errors

To our surprise, we found inconsistencies or type errors in all case studies except Berkeley DB. Berkeley DB is not relevant in this context, because it was already developed with CIDE and an early version of our type system; thus, we already eliminated all type errors in Berkeley DB during development. In all other case studies that were developed without a product-line–aware type system, we checked existing annotations in released source code.

In MobileMedia Release 6 (and Release 8), we found that a variant with SMS but without PHOTO would not compile. On closer inspection, we found that feature SMS actually depends on PHOTO, it is only meant to send photos, not music or video. This dependency was neither shown in the simplified feature model published in [Figueiredo et al. 2008], nor in a feature model provided by the authors on request, nor was any description about the relationship of features shipped with the source code. After adding this dependency to the feature model, CIDE indicates that all variants are well-typed. Detecting such mismatch between feature model and implementation is a typical example of the strength of product-line–aware type systems.

In Release 8, MobileMedia has five additional features, and annotations are more complex. CIDE initially indicated several type errors, because we inferred an incorrect feature model from the source code; we could easily fix this when we received a complete feature model from the authors and added the constraint between SMS and PHOTO as discussed above. Still, there were two remaining type errors caused by incorrectly annotated import statements (import statements are not part of the CFJ or the FJ calculus but are checked in CIDE). While the target class and its references were correctly annotated, two corresponding import statements were not annotated. This causes a Java type error in several variants when a removed class is imported (e.g., in variants with SMS but without CAPTUREPHOTO and without VIDEO, or in variants with COPYMEDIA but without PHOTO). The type system in CIDE can point out even such seemingly insignificant errors.

Also in Mobile RSS Reader, our type system found inconsistencies: Variants with both MIDP20 and SMALLMEM and variants with TESTUI but without MIDP10 contain type errors. Our domain knowledge is not sufficient to judge whether these are undocumented constraints or incorrect implementations. As an easy fix, adding the constraints '$\neg(\text{MIDP20} \land \text{SMALLMEM})$' and '$\text{TESTUI} \Rightarrow \text{MIDP10}$' reduces the number of possible variants, but then all variants are well-typed. It is up to the developers and domain experts to either change the implementation or the feature model.

Additionally, we found some fragments in Mobile RSS Reader that are never included in any variant (called dead feature code or zombie features [Tartler et al. 2009]). To include these code fragments, their annotations would require a feature selection that is not allowed by the feature model. Although, such dead-feature-code analysis is not part of the type system (dead feature code is always well-typed regarding reachability constraints), we can easily add a warning to our implementation to point out dead feature code.

Finally, in Lampiro, we already had difficulties to create a single Java version of the source code with all features (for backward compatibility). We found that feature SCREENSAVER is dead (since the first revision in the project's repository) and must never be selected: Its implementation

```
79    // #ifndef GLIDER
80    setTitle("Lampiro");
81    Image logo = Image.createImage("/icons/lampiro_icon.png");
82    UILabel ul = new UILabel("Loading Lampiro...");
83    // #endif
84    UILabel up = new UILabel(logo);
85    up.setAnchorPoint(Graphics.HCENTER | Graphics.VCENTER);
86    uvl.insert(up, 1, logo.getHeight()+10, UILayout.CONSTRAINT_PIXELS);
87
88    ul.setAnchorPoint(Graphics.HCENTER | Graphics.VCENTER);
89    uvl.insert(ul, 2, UIConfig.font_body.getHeight(), UILayout.CONSTRAINT_PIXELS);
```

Fig. 12.   Code excerpt from Lampiro *(SplashScreen.java)* with type errors when accessing local variables *logo* and *ul* in lines 84, 86, and 89 in variants with GLIDER.

Table II. Performance statistics of our case studies

Software product line	t_{Var} (sec)	t_{SPL} (sec)	#Variants	#Checks	#SATP	#USATP
MobileMedia (rel. 6)	0.2	1.3	144	5 714	1 924	39
MobileMedia (rel. 8)	0.3	1.8	2 784	7 359	3 569	111
Mobile RSS Reader	0.6	8.3	2 048	35 094	10 684	127
Lampiro	2.0	19.0	2 048	72 534	780	26
Berkeley DB	2.6	21.0	3.6 billion	70 316	19 517	324

t_{Var}: time to compile a single variant; t_{SPL}: time to evaluate all reachability checks; #Variants: approximate number of potential variants; #Checks: number of performed reachability checks; #SATP: number of SAT problems solved; #USATP: number of unique SAT problems solved

calls methods that do not exist, introduces duplicate methods, contains both missing and duplicate import declarations. Similarly, feature GLIDER is dead; it is obvious from code fragments as shown in Figure 12 that it makes no sense selecting this feature. Since GLIDER was only introduced in the last revision in the repository; we assume that it is an incomplete part of an upcoming feature. Our type system in CIDE points to these problems immediately. It forces developers to document in the feature model that certain features are incomplete and must not be selected.

All in all, we did not expect to find many errors, because all product lines released their code, and because the number of features is still manageable small. We were surprised to find small inconsistencies or type errors in every product line that was annotated with *#ifdef* directives. In all cases these were only minor problems (undocumented dependencies, forgotten annotation on an import statement, dead feature code), nothing significant and all easy to fix. Nevertheless, this shows how easy subtle errors can be introduced into well-developed product lines and how product-line–aware type systems can help to maintain consistency and fully document all implementation-relevant dependencies between features. In Berkeley DB, our type system helped to achieve consistency across the entire development process.

8.3. Performance

Finally, to provide some intuition about the complexity and performance of type checking a software product line, we measured the time to compile a single variant (t_{Var}) and the time to check all reachability constraints in the software product line (t_{SPL}).[20] Additionally, we estimated the number of variants to illustrate what it would mean to check every variant in isolation. Our current implementation of the type system is about ten times slower than Eclipse's industrial-strength compiler, that means type checking the entire product line takes as long as type checking ten variants (a fraction of the number of possible variants). Detailed results for all case studies are shown in Table II.

The slowdown is mostly caused by our algorithm to locate the pairs for reachability checks for method invocation, field access, type reference, and others, as described in the calculus. There are

---

[20]We measured all times on a standard 2.66 GHz lab PC with 4 GB RAM, Windows Vista, Sun Java VM 1.6.0.03, and Eclipse 3.5.

up to such 72 534 pairs in our case studies as shown in Table II. To enable quick *incremental* type checking on changes to the source code, to annotations, or to the feature model, we also store all checks for future reevaluation. We assume that an optimized implementation can significantly speed up this process. In contrast, the time needed to actually solve SAT problems is marginal. Many checks (60–98 %) can be skipped without consulting an SAT solver either (a) because neither code element is annotated or (b) because both are annotated with the identical feature expression. For the remaining checks, the results for unique feature combinations can be cached, so that, in our case studies, only some hundred unique SAT problems remain to be solved. Solving all SAT problems requires less than 50 ms for each product line.

This shows that, although reachability checks are required in all typing judgments, they can be executed with reasonable performance that is acceptable for practical development. Our current implementation slows down type checking by a factor of ten, which means that for every product line with more than ten potential variants, it is faster to check the entire product line during domain engineering than to check every variant in isolation during application engineering. Type checking is still reasonably fast that it can be executed in the background during development to find errors as early as possible.

9. RELATED WORK

Type checking product lines

The idea of type checking an entire software product lines (instead of individual variants) emerged from research on generative programming.

First, in an influential approach, Huang et al. [2005] ensure that Java code generated by their tool *SafeGen* is well-typed. Though their tool is used for metaprogramming in general, not as product-line technology, the basic idea is similar to our theorem generation preserves typing. Since there is no need for backward compatibility, alternative features are supported natively. Using first-order logics and theorem provers, they check whether generators written in their confined metalanguage (with selection and iteration operators) produce well-typed output for arbitrary Java input. However, checks cover only some of Java's typing rules, i.e., there is no *guarantee* that the output is well-typed. In recent work, they introduced a newer metaprogramming language *MorphJ* with similar constructs that supports modular type checking and has been proven type sound [Huang and Smaragdakis 2010].

The work on checking the generation mechanism instead of individual input programs in *Safe-Gen* influenced Czarnecki and Pietroszek [2006] to check an entire product line instead of individual variants. Specifically, they target product lines of UML models in their tool *fmp2rsm* and guarantee well-formedness for all variants. In earlier work, Czarnecki and Antkiewicz [2005] implemented a tool environment to develop a product line of UML models, very similar to CIDE: they extended an existing UML editor such that a user can annotate *presence conditions* to UML elements like classes or associations; a variant of the UML model is generated by removing elements of which the annotation evaluates to *false* for a feature selection. In this environment, also backward compatibility to the existing UML editor was implicit. Czarnecki and Pietroszek [2006] then describe a mechanism for this tool environment to check that all variants conform to certain well-formedness rules of UML – e.g., 'an association in UML class diagrams connects exactly two elements'. These well-formedness rules are similar to typing rules in programming languages and can be specified in UML's metamodel formally (and machine readable) using constraints written in the *Object Constraint Language (OCL)*. Their tool transforms presence conditions, the feature model, and OCL constraints into a propositional formula, which can be solved by an off-the-shelf SAT solver in a single step. Error messages are reconstructed from the SAT solver's result. Well-formedness can only be guaranteed against those constraints that have been specified (machine-readable) with OCL. For UML those must be first inferred from the informal, textual UML specification, which is similar to how Java's typing rules must be inferred from the textual Java Language Specification. The authors do not discuss completeness of their inferred OCL constraints. The metaexpression solution

for alternative features was first described for their tool [Czarnecki and Antkiewicz 2005]; however, metaexpressions have not (yet) been considered in their well-formedness checks [Czarnecki and Pietroszek 2006].

Beyond annotations on existing languages, there have been approaches to type check product lines written in specialized architectures or with specialized languages using constructs such as aspects, class refinements, or mixins. These approaches generate variants by composing code modules. In some sense, feature annotations and feature composition are two sides of the same coin: one removes code from a common base (sometimes called negative variability), the other merges already separated code (positive variability) [Kästner et al. 2008]. Both approaches can be used (also in combination) to implement product lines. Refactoring between implementations based on annotations and based on composition is usually possible [Kästner et al. 2009]; hence, within the boundaries caused by alternative features, it is also possible to use a composition-based type system indirectly for annotation-based implementations and vice versa.

Some compositional approaches can check feature modules in isolation, so only their combination into variants need to be checked; separate checking is possible for architectures with separately compiled components or plug-ins, as well as for several specialized languages, e.g., [Ossher and Tarr 2000; McDirmid et al. 2001; Warth et al. 2006; Chae and Blume 2008; Hutchins 2009; Bettini et al. 2010; Apel and Hutchins 2010]. In some scenarios, architectures are possible, in which all features are independent plug-ins that can be combined without type conflicts [Chae and Blume 2008]. In many languages, by analyzing module interfaces, we could derive dependencies that describe which modules can be combined together; we could either use such dependencies to extract an (implementation-specific) feature model [She et al. 2011] or consider these dependencies themselves as feature model. Nevertheless, in the product-line community, feature models often describe domain knowledge beyond just implementation dependencies. Thus, we typically need an extra step to checking actual variability in the implementation against the intended variability described in the feature model [Metzger et al. 2007; Thaker et al. 2007].

The first approach of type checking *all* valid variants (intended variability) of a product line implemented by feature composition was *safe composition* by Thaker et al. [2007]. They analyze language semantics of Jak [Batory et al. 2004], a Java dialect that supports mixin-style class refinements (including native support for alternative features). To check types, they identify six constraints that need to be satisfied, which their tool maps to propositional formulas and checks with an SAT solver. One constraint deals with references to fields and methods (roughly corresponding to T-FIELD and T-INVK), two deal with abstract classes and interfaces (no correspondence in Featherweight Java), and three deal with specific constructs of the Jak composition mechanism (no correspondence in Featherweight Java). Their checks are not claimed or even proved complete, and in fact – compared to CFJ – checks that ensure the presence of types uses in signatures are missing, e.g., (M.1), (M.3). In recent work, an extension of safe composition was eventually also formalized and proved type-sound with a machine-checked model by providing an algorithm to reduce it to Lightweight Java [Delaware et al. 2009].

In a parallel line of research, we have formalized a calculus *Feature Featherweight Java (FFJ)* for class refinement and module composition [Apel et al. 2008] and extended it toward checking entire product lines as FFJ_{PL} [Apel et al. 2010]. In this work, we entirely drop backward compatibility since the host language with its composition semantics is already incompatible to Java and there is no sophisticated tool support, yet. Instead, we aimed at flexibility so that even alternative classes with different supertypes, or alternative fields with different types and alternative methods with different return types are possible. Compared to CFJ the formalization is much more complex, because a term in the product line may have different types and even the subtype relation may change in different variants depending on the feature selection. In the worst case, type checking has exponential complexity. CFJ and FFJ tackle type checking software product lines for different implementation mechanisms and from different perspectives: CFJ targets at annotations and tool support focusing on developers while FFJ targets module composition and explores maximum flexibility.

Conditional language constructs

Independent of product-line research, the programming language community developed several type systems that support type conditions on methods or other language constructs. So, invoking a conditional method is only well-typed when the condition is satisfied in the context of the invocation. Conditional language constructs are discussed in the context of parametric polymorphism. For example, in a collection class, such as *List*, clients should only be allowed to invoke a method *print* if the class is parametrized with a type that can be printed; a collection should only implement the interface *Printable* if the type parameter implements this interface as well. Conditional language constructs have been explored in object-oriented languages at least since CLU [Liskov et al. 1981] and have been studied, for example, in extensions to Cecil [Litvinov 1998], Java [Myers et al. 1997; Huang et al. 2007], and C# [Emir et al. 2006]. In all these languages, type constraints are structural constraints (parameter X contains method Y) or subtyping constraints (parameter X is a subtype/supertype of Y).

Conditional methods with type constraints and CFJ are related, because both restrict the access to methods in some variants (#ifdef vs. condition on type parameter) and both statically ensure that all variants are well-typed. So, in some sense, we could replace *#ifdef* directives on statements by conditions on type parameters and instead of generating a variant by removing code, we could instantiate the program with a suitable type parameter. However, there are four important differences:

- *Code removal vs. multiple instances.* Our work addresses conditional compilation in the context of product lines, such that code is actually removed in a generation step. In contrast, all languages with conditional methods we are aware of do not generate variants but check that a present method is never called when the condition on the type parameter evaluates to *false*. Type conditions have the benefit that different instances of a class with different configurations may be used in the same program, but they does not remove code and thus does not reduce binary size as sometimes desired in product-line development, especially for embedded systems [Beuche et al. 1999; Lohmann et al. 2006; Rosenmüller et al. 2009].
- *Expressiveness of conditions.* Compared to a full feature model, the expressiveness of type conditions is restricted. In languages with structural constraints, they can express part-of relationships; in language with subtyping constraints, they can express simple parent-child relationships (similar to our initial 'set relations' implementation, see Sec. 7). Most type conditions have the benefit that reasoning can be performed without a SAT solver; however, more expressive feature constraints are needed in product-line practice (see Sec. 8), such as alternative features, negated features ($\neg A$), or propositional expressions (e.g., $A \vee \neg B \wedge C$).
- *Granularity.* Annotations and type conditions provide different levels of granularity. In contemporary languages with type conditions, typically conditions can only be placed on methods (and sometimes fields and supertypes); type conditions aim primarily at providing flexible libraries. In contrast, *#ifdef* directives and annotations in CFJ and CIDE are more flexible and can annotate entire classes, individual statements, or even method parameters, which is typically not needed in libraries. Our work targets at variability in applications and product lines, in which also the behavior of an individual method may change depending on the feature selection. Of the four examples in Section 3, only the first can be implemented and checked with type conditions of contemporary languages.
- *Backward compatibility.* Finally, to add type conditions to Cecil, Java, or C#, all approaches introduce new language constructs. In contrast, we aim explicitly at backward compatibility to reuse the existing tool infrastructure.

These differences are mostly design decisions for a specific language. It is possible to develop conditional language constructs that are similar to CFJ (backward compatible, at finer granularity, with more expressive conditions) or product-line–aware type systems with characteristics of conditional language constructs. However, so far the product-line community and the programming language community pursued different goals (product-line development by code removal, backward compat-

ibility, flexible annotations, and alternative implementations vs. expressive type system for libraries and multiple instances), which lead to different design decisions. With contemporary conditional compilation constructs, our case studies would be very difficult to implement. We argue that both approaches are complementary and may eventually converge. In this context, we contribute a different perspective with different design decisions and their trade-offs for conditional language constructs.

10. CONCLUSION

We have formally discussed a type system for an entire software product line that is implemented with annotations on a common code base. Instead of checking all – possibly millions – of variants that can be generated from a product line in isolation, we check the product line itself and *guarantee* that all variants generated from a well-typed product line are well-typed. We have shown that CFJ can be modeled in a backward compatible fashion on top of FJ, extending only the typing rules and auxiliary functions with local checks on annotations.

The formalization was motivated by our product-line tool CIDE for Java and other languages. Though, CFJ (or FJ) covers only a small excerpt from the Java specification, the formalization provides several insights on how to design a product-line–aware type system, such as the concept of reachability conditions, the theorem generation preserves typing, and the design decision of backward compatibility. With the small scope it also allowed to explore the implementation of alternative features with metaexpressions in detail.

In four case studies, we have shown that type checking an entire software product line is feasible, useful, and reasonably fast. With our implementation in CIDE, we even found inconsistencies (undocumented dependencies, forgotten annotations on an import statements, dead feature code) in all analyzed product lines that were developed with *#ifdef* directives. With a product-line–aware type system, we detect such problems already early during product-line development, instead of late when problematic variant with a specific feature combination is eventually compiled. Although these product lines contain hundreds of annotations, sometimes with at a fine level of granularity or with complex or nested feature expressions, we can efficiently automate reachability checking.

In future work, we intend to explore paths toward extensions for other code and non-code languages and their interactions (inter-language typing). Furthermore, we intend to apply verification and validation tools to entire product lines to find also semantic errors. Our long term goal is to provide a language-independent product-line tool that counteracts the inherent complexity of product lines by detecting possible errors as early as possible.

REFERENCES

APEL, S. AND HUTCHINS, D. 2010. A calculus for uniform feature composition. *ACM Trans. Program. Lang. Syst. (TOPLAS) 32,* 5, 1–33.

APEL, S., KÄSTNER, C., GRÖSSLINGER, AND LENGAUER, C. 2010. Type safety for feature-oriented product lines. *Automated Software Engineering 17,* 3, 251–300.

APEL, S., KÄSTNER, C., AND LENGAUER, C. 2008. Feature Featherweight Java: A calculus for feature-oriented programming and stepwise refinement. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE).* ACM Press, New York, 101–112.

ATKINS, D. L., BALL, T., GRAVES, T. L., AND MOCKUS, A. 2002. Using version control data to evaluate the impact of software tools: A case study of the Version Editor. *IEEE Trans. Softw. Eng. (TSE) 28,* 7, 625–637.

BASS, L., CLEMENTS, P., AND KAZMAN, R. 1998. *Software Architecture in Practice.* Addison-Wesley, Boston, MA.

BATORY, D. 2005. Feature models, grammars, and propositional formulas. In *Proc. Int'l Software Product Line Conference (SPLC).* Lecture Notes in Computer Science Series, vol. 3714. Springer-Verlag, Berlin/Heidelberg, 7–20.

BATORY, D. AND GERACI, B. J. 1997. Composition validation and subjectivity in GenVoca generators. *IEEE Trans. Softw. Eng. (TSE) 23,* 2, 67–82.

BATORY, D., SARVELA, J. N., AND RAUSCHMAYER, A. 2004. Scaling step-wise refinement. *IEEE Trans. Softw. Eng. (TSE) 30,* 6, 355–371.

BENAVIDES, D., TRINIDAD, P., AND RUIZ-CORTES, A. 2005. Automated reasoning on feature models. In *Proc. Conf. Advanced Information Systems Engineering (CAiSE).* Lecture Notes in Computer Science Series, vol. 3520. Springer-Verlag, Berlin/Heidelberg, 491–503.

BERGEL, A., DUCASSE, S., AND NIERSTRASZ, O. 2005. Classbox/J: Controlling the scope of change in Java. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM Press, New York, 177–189.

BETTINI, L., DAMIANI, F., AND SCHAEFER, I. 2010. Implementing software product lines using traits. In *Proc. Symp. Applied Computing (SAC).* ACM Press, New York, 2098–2104.

BEUCHE, D., GUERROUAT, A., PAPAJEWSKI, H., SCHRÖDER-PREIKSCHAT, W., SPINCZYK, O., AND SPINCZYK, U. 1999. The PURE family of object-oriented operating systems for deeply embedded systems. In *Int'l Symp. Object-Oriented Real-Time Distributed Computing (ISORC).* IEEE Computer Society, Los Alamitos, CA, 45–53.

BEUCHE, D., PAPAJEWSKI, H., AND SCHRÖDER-PREIKSCHAT, W. 2004. Variability management with feature models. *Sci. Comput. Program. 53,* 3, 333–352.

BRACHA, G. AND COOK, W. 1990. Mixin-based inheritance. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA).* ACM Press, New York, 303–311.

CHAE, W. AND BLUME, M. 2008. Building a family of compilers. In *Proc. Int'l Software Product Line Conference (SPLC).* IEEE Computer Society, Los Alamitos, CA, 307–316.

CHAPMAN, M. 2006. Extending JDT to support Java-like languages. Invited Talk at EclipseCon'06.

CONEJERO, J. M., FIGUEIREDO, E., GARCIA, A., HERNÁNDEZ, J., AND JURADO, E. 2009. Early crosscutting metrics as predictors of software instability. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE).* Lecture Notes in Business Information Processing Series, vol. 33. Springer-Verlag, Berlin/Heidelberg, 136–156.

CZARNECKI, K. AND ANTKIEWICZ, M. 2005. Mapping features to models: A template approach based on superimposed variants. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE).* Lecture Notes in Computer Science Series, vol. 3676. Springer-Verlag, Berlin/Heidelberg, 422–437.

CZARNECKI, K. AND EISENECKER, U. 2000. *Generative Programming: Methods, Tools, and Applications.* ACM Press/Addison-Wesley, New York.

CZARNECKI, K. AND PIETROSZEK, K. 2006. Verifying feature-based model templates against well-formedness OCL constraints. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE).* ACM Press, New York, 211–220.

DELAWARE, B., COOK, W. R., AND BATORY, D. 2009. Fitting the pieces together: A machine-checked model of safe composition. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE).* ACM Press, New York, 243–252.

DUCASSE, S., NIERSTRASZ, O., SCHÄRLI, N., WUYTS, R., AND BLACK, A. P. 2006. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst. (TOPLAS) 28,* 2, 331–388.

EMIR, B., KENNEDY, A., RUSSO, C., AND YU, D. 2006. Variance and generalized constraints for C[#] generics. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP).* Lecture Notes in Computer Science Series, vol. 4067. Springer-Verlag, Berlin/Heidelberg, 279–303.

ERNST, M., BADROS, G., AND NOTKIN, D. 2002. An empirical analysis of C preprocessor use. *IEEE Trans. Softw. Eng. (TSE) 28,* 12, 1146–1170.

FAVRE, J.-M. 1997. Understanding-in-the-large. In *Proc. Int'l Workshop on Program Comprehension.* IEEE Computer Society, Los Alamitos, CA, 29.

FIGUEIREDO, E. ET AL. 2008. Evolving software product lines with aspects: An empirical study on design stability. In *Proc. Int'l Conf. Software Engineering (ICSE).* ACM Press, New York, 261–270.

FLATT, M., KRISHNAMURTHI, S., AND FELLEISEN, M. 1998. Classes and mixins. In *Proc. Symp. Principles of Programming Languages (POPL).* ACM Press, New York, 171–183.

FUHRER, R. M., KELLER, M., AND KIEŻUN, A. 2007. Advanced refactoring in the Eclipse JDT: Past, present, and future. In *Proc. ECOOP Workshop on Refactoring Tools (WRT).* TU Berlin, Berlin, 30–31.

GANESAN, D., LINDVALL, M., ACKERMANN, C., MCCOMAS, D., AND BARTHOLOMEW, M. 2009. Verifying architectural design rules of the flight software product line. In *Proc. Int'l Software Product Line Conference (SPLC).* Carnegie Mellon University, Pittsburgh, PA, 161–170.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2005. *Java[TM] Language Specification* 3rd Ed. The Java[TM] Series. Addison-Wesley, Amsterdam.

GRECHANIK, M., BATORY, D., AND PERRY, D. 2004. Design of large-scale polylingual systems. In *Proc. Int'l Conf. Software Engineering (ICSE).* IEEE Computer Society, Washington, DC, 357–366.

HUANG, S. S. AND SMARAGDAKIS, Y. 2010. Morphing: Structurally shaping a class by reflecting on others. *ACM Trans. Program. Lang. Syst. (TOPLAS).* to appear.

HUANG, S. S., ZOOK, D., AND SMARAGDAKIS, Y. 2005. Statically safe program generation with SafeGen. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. Lecture Notes in Computer Science Series, vol. 3676. Springer-Verlag, Berlin/Heidelberg, 309–326.

HUANG, S. S., ZOOK, D., AND SMARAGDAKIS, Y. 2007. cJ: Enhancing Java with safe type conditions. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*. ACM Press, New York, 185–198.

HUTCHINS, D. 2009. Pure subtype systems: A type theory for extensible software. Ph.D. thesis, University of Edinburgh.

IGARASHI, A., PIERCE, B., AND WADLER, P. 2001. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst. (TOPLAS) 23,* 3, 396–450.

JEPSEN, H. P. AND BEUCHE, D. 2009. Running a software product line – standing still is going backwards. In *Proc. Int'l Software Product Line Conference (SPLC)*. Carnegie Mellon University, Pittsburgh, PA, 101–110.

JOHNSON, R. E. AND FOOTE, B. 1988. Designing reusable classes. *Journal of Object-Oriented Programming (JOOP) 1,* 2, 22–35.

KANG, K., COHEN, S. G., HESS, J. A., NOVAK, W. E., AND PETERSON, A. S. 1990. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Tech. Rep. CMU/SEI-90-TR-21, SEI, Pittsburgh, PA.

KÄSTNER, C. AND APEL, S. 2008. Type-checking software product lines – A formal approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*. IEEE Computer Society, Los Alamitos, CA, 258–267.

KÄSTNER, C. AND APEL, S. 2009. Virtual separation of concerns – A second chance for preprocessors. *Journal of Object Technology (JOT) 8,* 6, 59–78.

KÄSTNER, C., APEL, S., AND BATORY, D. 2007. A case study implementing features using AspectJ. In *Proc. Int'l Software Product Line Conference (SPLC)*. IEEE Computer Society, Los Alamitos, CA, 223–232.

KÄSTNER, C., APEL, S., AND KUHLEMANN, M. 2008. Granularity in software product lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM Press, New York, 311–320.

KÄSTNER, C., APEL, S., AND KUHLEMANN, M. 2009. A model of refactoring physically and virtually separated features. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM Press, New York, 157–166.

KÄSTNER, C., APEL, S., TRUJILLO, S., KUHLEMANN, M., AND BATORY, D. 2009. Guaranteeing syntactic correctness for all product line variants: A language-independent approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*. Lecture Notes in Business Information Processing Series, vol. 33. Springer-Verlag, Berlin/Heidelberg, 175–194.

KÄSTNER, C., THÜM, T., SAAKE, G., FEIGENSPAN, J., LEICH, T., WIELGORZ, F., AND APEL, S. 2009. FeatureIDE: Tool framework for feature-oriented software development. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, Washington, DC, 611–614.

KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science Series, vol. 1241. Springer-Verlag, Berlin/Heidelberg, 220–242.

KRONE, M. AND SNELTING, G. 1994. On the inference of configuration structures from source code. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, 49–57.

KRUEGER, C. W. 2002. Easing the transition to software mass customization. In *Proc. Int'l Workshop on Software Product-Family Engineering (PFE)*. Lecture Notes in Computer Science Series, vol. 2290. Springer-Verlag, Berlin/Heidelberg, 282–293.

LEICH, T., APEL, S., AND MARNITZ, L. 2005. Tool support for feature-oriented software development: FeatureIDE: An eclipse-based approach. In *Proc. OOPSLA Workshop on Eclipse Technology eXchange (ETX)*. ACM Press, New York, 55–59.

LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, J. C., SCHEIFLER, R., AND SNYDER, A. 1981. *CLU Reference Manual*. Lecture Notes in Computer Science Series, vol. 114. Springer-Verlag, Berlin/Heidelberg.

LITVINOV, V. 1998. Contraint-based polymorphism in Cecil: Towards a practical and static type system. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, New York, 388–411.

LOHMANN, D., SCHELER, F., TARTLER, R., SPINCZYK, O., AND SCHRÖDER-PREIKSCHAT, W. 2006. A quantitative analysis of aspects in the eCos kernel. *ACM SIGOPS Operating Systems Review 40,* 4, 191–204.

MCDIRMID, S., FLATT, M., AND HSIEH, W. 2001. Jiazzi: New-age components for old-fashioned Java. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, New York, 211–222.

MCDIRMID, S. AND ODERSKY, M. 2006. The Scala plugin for Eclipse. In *Proc. ECOOP Workshop on Eclipse Technology eXchange (ETX)*. published online `http://atlanmod.emn.fr/www/papers/eTX2006/`.

MENDONÇA, M., WĄSOWSKI, A., AND CZARNECKI, K. 2009. SAT-based analysis of feature models is easy. In *Proc. Int'l Software Product Line Conference (SPLC)*. Carnegie Mellon University, Pittsburgh, PA, 231–240.

METZGER, A., POHL, K., HEYMANS, P., SCHOBBENS, P.-Y., AND SAVAL, G. 2007. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proc. Int'l Requirements Engineering Conf. (RE)*. IEEE Computer Society, Los Alamitos, CA, 243–253.

MYERS, A. C., BANK, J. A., AND LISKOV, B. 1997. Parameterized types for Java. In *Proc. Symp. Principles of Programming Languages (POPL)*. ACM Press, New York, 132–145.

NIPKOW, T. AND VON OHEIMB, D. 1998. Java light is type-safe – definitely. In *Proc. Symp. Principles of Programming Languages (POPL)*. ACM Press, New York, 161–170.

OSGI ALLIANCE. 2009. *OSGi Service Platform Core Specification* Release 4, Version 4.2 Ed. `http://www.osgi.org`.

OSSHER, H. AND TARR, P. 2000. Hyper/J: Multi-dimensional separation of concerns for Java. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM Press, New York, 734–737.

PEARSE, T. T. AND OMAN, P. W. 1997. Experiences developing and maintaining software in a multi-platform environment. In *Proc. Int'l Conf. Software Maintenance (ICSM)*. IEEE Computer Society, Los Alamitos, CA, 270–277.

PIERCE, B. C. 2002. *Types and Programming Languages*. MIT Press, Cambridge, MA.

POHL, K., BÖCKLE, G., AND VAN DER LINDEN, F. J. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, Berlin/Heidelberg.

PREHOFER, C. 1997. Feature-oriented programming: A fresh look at objects. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*. Lecture Notes in Computer Science Series, vol. 1241. Springer-Verlag, Berlin/Heidelberg, 419–443.

ROSENMÜLLER, M., APEL, S., LEICH, T., AND SAAKE, G. 2009. Tailor-made data management for embedded systems: A case study on Berkeley DB. *Data and Knowledge Engineering (DKE) 68,* 12, 1493–1512.

ROSENTHAL, M. 2009. Alternative features in Colored Featherweight Java. M.S. thesis, University of Passau.

SCHOBBENS, P.-Y., HEYMANS, P., AND TRIGAUX, J.-C. 2006. Feature diagrams: A survey and a formal semantics. In *Proc. Int'l Requirements Engineering Conf. (RE)*. IEEE Computer Society, Los Alamitos, CA, 139–148.

SHE, S., LOTUFO, R., BERGER, T., WĄSOWSKI, A., AND CZARNECKI, K. 2011. Reverse engineering feature models. In *Proc. Int'l Conf. Software Engineering (ICSE)*. ACM Press, New York. to appear.

SHE, S., LOTUFO, R., BERGER, T., WĄSOWSKI, A., AND CZARNECKI, K. 2010. The variability model of the Linux kernel. In *Proc. Int'l Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*. University of Duisburg-Essen, Essen, 45–51.

SPENCER, H. AND COLLYER, G. 1992. #ifdef considered harmful or portability experience with C news. In *Proc. USENIX Conf*. USENIX Association, Berkeley, CA, 185–198.

STRNIŠA, R., SEWELL, P., AND PARKINSON, M. 2007. The Java module system: Core design and semantic definition. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, New York, 499–514.

SZYPERSKI, C. 1997. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, Boston, MA.

TARR, P., OSSHER, H., HARRISON, W., AND SUTTON, JR., S. M. 1999. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, Los Alamitos, CA, 107–119.

TARTLER, R., SINCERO, J., SCHRÖDER-PREIKSCHAT, W., AND LOHMANN, D. 2009. Dead or alive: Finding zombie features in the Linux kernel. In *Proc. GPCE Workshop on Feature-Oriented Software Development (FOSD)*. ACM Press, New York, 81–86.

THAKER, S., BATORY, D., KITCHIN, D., AND COOK, W. 2007. Safe composition of product lines. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*. ACM Press, New York, 95–104.

THÜM, T. 2010. test, University of Magdeburg.

THÜM, T., BATORY, D., AND KÄSTNER, C. 2009. Reasoning about edits to feature models. In *Proc. Int'l Conf. Software Engineering (ICSE)*. IEEE Computer Society, Washington, DC, 254–264.

WARTH, A., STANOJEVIĆ, M., AND MILLSTEIN, T. 2006. Statically scoped object adaptation with expanders. In *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. ACM Press, New York, 37–56.

WRIGHT, A. K. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Information and computation 115,* 1, 38–94.