Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach

Christian Kästner[1], Sven Apel[2], Salvador Trujillo[3], Martin Kuhlemann[1], and Don Batory[4]

[1] School of Computer Science, University of Magdeburg, `ckaestne/mkuhlema@ovgu.de`
[2] Dept. of Informatics and Math., University of Passau, `apel@uni-passau.de`
[3] IKERLAN Research Centre, Mondragon, Spain, `STrujillo@ikerlan.es`
[4] Dept. of Computer Science, University of Texas at Austin, `batory@cs.utexas.edu`

Abstract. A *software product line (SPL)* is a family of related program variants in a well-defined domain, generated from a set of features. A fundamental difference from classical application development is that engineers develop not a single program but a whole family with hundreds to millions of variants. This makes it infeasible to separately check every distinct variant for errors. Still engineers want guarantees on the entire SPL. A further challenge is that an SPL may contain artifacts in different languages (code, documentation, models, etc.) that should be checked. In this paper, we present CIDE, an SPL development tool that guarantees syntactic correctness for all variants of an SPL. We show how CIDE's underlying mechanism abstracts from textual representation and we *generalize* it to arbitrary languages. Furthermore, we *automate* the generation of plug-ins for additional languages from annotated grammars. To demonstrate the language-independent capabilities, we applied CIDE to a series of case studies with artifacts written in Java, C++, C, Haskell, ANTLR, HTML, and XML.

1 Introduction

A *Software Product Line (SPL)* is a set of software-intensive systems that shares a common, managed set of features, satisfying the specific needs of a domain [7]. A *feature* is an end-user visible requirement that is used to describe commonalities or differences in this domain [20, 27]. Different programs of the SPL, called *variants*, can be generated from a common code base by combining features. Already with a few features, a high number of distinct variants can be generated [27].

The ability to generate many variants is beneficial because variants can be tailored to specific scenarios according to customer requirements. At the same time, this raises new challenges: Traditional application development focuses on the design, implementation, and testing of a single program; detecting errors in an SPL is difficult as an error may appear only in some variants with specific features or feature combinations [28, 9]. Because of the sheer number of variants (already with 33 independent optional features, there is a distinct variant for every person on the planet), generating, separately compiling every variant in isolation is usually infeasible. Thus, novel techniques for *checking the entire SPL* instead of *checking each variant separately* are needed [9, 35].

Approaches for checking SPLs are especially challenging considering the fact that an SPL typically contains artifacts written in different languages. Beyond source code, an SPL can also contain non-code artifacts as build scripts, models, and documentation. SPL tools should handle different artifact types in a uniform way (a.k.a. principle of uniformity) [3]. While some SPL implementation mechanisms (preprocessors [27], XVCL [18], Gears [26], pure::variants [4], etc.) are so general that they work on plain text files, even simple errors can go undetected. Even syntax errors in the target language – like missing a closing bracket – can be difficult to detect if the error occurs only in few variants. On the other hand, more sophisticated tools and languages that can detect certain errors in SPLs [3, 9, 24, 8, 16, 21] are usually available only for a single language.

In this paper, we present a tool called *Colored Integrated Development Environment (CIDE)* for SPL development that guarantees syntactical correctness for all variants and multiple languages. CIDE is similar to *#ifdef*-preprocessors, but by abstracting from plain text files and considering internal structure, simple rules can prevent syntax errors in a language-independent way. However, while a previous version of CIDE presented in [22] (with focus on expressiveness of language constructs) supported only Java, in this work, we *generalize* CIDE to a wide variety of languages. We extract underlying principles necessary for correctness in Java and propose a language-independent model that can be used for other languages as well. Finally, to minimize human effort for using this model with a new language, we *automate* the generation of language plug-ins for CIDE based on a grammar file of the target language.

Detecting syntax errors is only a first step in our endeavor to ensure a language-independent safe implementation of SPLs, which detects errors as early as possible in the development process. With this paper, we contribute a foundation for this endeavor: simple mechanisms can prevent common and difficult to find syntax errors, independent of the used language. On this foundation, further mechanisms to detect type errors [21, 25] or for verification [36] and model-checking [30] (which all require syntactically correct code and so far exist for single specific languages only) can be added for more detailed checks and multiple languages in future steps.

In order to demonstrate the practicality of our approach, we generated a number of plug-ins for (among others) the following code and non-code languages: Java, C, C++, C#, Haskell, JavaScript, ANTLR, HTML, XML. Subsequently, we used CIDE in seven small to medium-sized SPL projects written with these languages. In all projects, CIDE guarantees syntactical correctness of all generated variants in all languages. CIDE together with all languages and case studies presented in this paper can be downloaded from the project's web site: `http://fosd.de/cide/`.

2 Taxonomy of Errors in Software Product Lines & Related Work

There are many possible errors that can occur in an SPL's implementation. We provide a taxonomy to explain the problems we are addressing and to distinguish our approach from related work. In Figure 1, we give an overview and set the focus of this paper (highlighted boxes).

First, we classify three *kinds of errors*: syntactic errors, type errors and semantic errors (first line in Fig. 1). Syntax errors occur when a variant is ill-formed regarding the language's syntax, for example when an opened bracket is not closed. Type errors occur when the variant is ill-formed regarding the language's type system, e.g., a statement

Kind of Error	Syntax	Typing	Semantic	
Error Detection	Check Variants	Check SPL		
Languages	Single	Multiple	Inter-Language	
Implementation	Annotative	Compositional	Generators	...

Fig. 1. Taxonomy of errors and corresponding checks in SPLs (morphological box)

invoking a method that is not defined in that variant. Finally, semantic errors occur when the variant behaves incorrectly according to some (formal or informal) specification, and are the most difficult to detect. In this paper, we begin with syntax errors but give an outlook on detecting typing and semantic errors.

Second, we distinguish two *error detection approaches*: check variants or check the SPL itself (second line in Fig. 1). In the first case, not the SPL itself but (some or all) generated variants are checked [28]. A brute force strategy of generating and checking all variants is usually infeasible, because already with few features the number of variants that can be generated from an SPL explodes (for n independent optional features, there are 2^n distinct variants). This typically means that only some sampled variants are checked. In contrast, some approaches check the entire SPL itself [9, 35, 21] and guarantee correctness for all variants when this check passes. In our work, we want to check the SPL, not each variant separately.

Third, we classify checks by their coverage of different *programming languages*: single language, multiple languages, and inter-language errors. Some checks are specific to a single language. For example, different languages require different type checks. Next, there are errors that can occur in different languages and can be addressed by the same tool. Finally, there are errors that occur only at the interaction of multiple languages, e.g., the interface specification of a web service in a WSDL file and its implementation might not match. In our work, we focus on a mechanism that is language-independent.

Fourth, we distinguish checks by the general *implementation mechanism* of the SPL, which is usually put on top of a programming language. Although there are many different mechanisms, for brevity, our taxonomy considers only three groups. In annotative approaches – common in industry – code is annotated and removed for variant generation; typical examples include *'#ifdef'* preprocessor directives, Frames/XVCL [18], and commercial SPL tools like *Gears* [26] and *pure::variants* [4]. In contrast, compositional approaches – favored in academia – implement features in physically separated modules and compose them to generate variants; examples include frameworks [19] and different forms of components, aspects or feature modules [27, 3, 24, 2, 1]. Furthermore, several other implementation mechanisms like generators [8, 16] or version control systems [33] exist. All approaches have different advantages and disadvantages, e.g., regarding SPL adoption or expressiveness as discussed in [6] and [22], which justifies research on error detection for all of them. In this work, we focus on annotative approaches.

Related Work by Kind of Error. There is a large body of research on checking SPLs for errors. A first group of approaches focus on SPL *testing*, i.e. detecting *semantic errors* by running test cases [34, 28, 27]. Testing can be applied to different implementation mechanisms and different languages and can detect even inter-language defects, but only variants are tested, not the entire SPL. Another approach to detect semantic errors is to apply formal methods. Earlier work suggested specific languages to *verify* SPLs for compositional approaches [36, 29] or to check an SPL with annotations using *model checking* [30], but both have yet to show how they scale for real world SPLs.

Also for *type errors*, there has been effort to switch from type-checking individual variants to type-checking the entire SPL. Approaches exist for individual languages with annotative [21, 9, 25], compositional [35, 10] and generative [16, 17] implementations. Although suggested as a future extension in [9] and [35], to the best of our knowledge, approaches that cover multiple languages or inter-language typing have not been applied to SPL checking.

Finally, regarding *syntax errors*, compositional approaches separate feature modules physically so that the SPL can be checked by checking all features in isolation. Recent composition tools that support multiple languages in a uniform way [3, 1] can even check the SPL's syntax for multiple languages. For (language-specific) generators there have been approaches to check the generator, to ensure syntactical correct output for any input [16]. In contrast, annotative approaches (all features located in the same annotated code base) are typically so general that they work only on plain text, and detecting syntax errors is largely unexplored.

Syntactic Correctness in Annotative Approaches. Consider the code fragment in Figure 2 that shows a fragment of C code from Oracle's Berkeley DB[1] that uses C's preprocessor with multiple (partly nested) *'#ifdef'* annotations to generate different variants. Already, this short code fragment illustrates the complexity that may occur when implementing variability in SPLs, but more importantly, it illustrates a subtle error, which we deliberately introduced. Note that the opening curly bracket in Line 4 is only closed in Line 17 when the feature *HAVE_QUEUE* is selected, while other variants contain a syntax error. Although this example may appear trivial, it is a matter of scale. In our projects (see Sec. 5), we experienced such problems frequently and it was often not obvious to find their cause. In large SPLs with many features, syntax errors can easily occur in some variants and can be difficult to detect, especially when located in nested annotations and thus only in very few variants [32].

```
 1 static int __rep_queue_filedone(
       dbenv, rep, rfp)
 2   DB_ENV *dbenv;
 3   REP *rep;
 4   __rep_fileinfo_args *rfp; {
 5 #ifndef HAVE_QUEUE
 6   COMPQUIET(rep, NULL);
 7   COMPQUIET(rfp, NULL);
 8   return (__db_no_queue_am(dbenv));
 9 #else
10   db_pgno_t first, last;
11   u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16   // over 100 lines of additional code
17 }
18 #endif
```

Fig. 2. Code excerpt of Berkeley DB, with syntax error in variants without *HAVE_QUEUE*.

Interestingly, there are some annotative tools that can guarantee syntactic correctness for some languages by transforming and annotating software artifacts on a higher level of abstraction. One example is Czarnecki's tool *fmp2rsm* [9] to generate variants of annotated UML models. Using this tool, syntax errors (e.g., a class without a name) cannot occur because annotations and variant generation is not performed on the textual

[1] `http://www.oracle.com/database/berkeley-db`

representation of the model, but on an abstract level with the Rational Software Modeler engine, which does not allow transformations that would invalidate UML syntax. In other fields of software engineering, this abstraction principle is also frequently applied. For example, refactorings in IDEs such as Eclipse are usually not performed directly on the textual source code, but on an abstract representation like an abstract syntax tree [12]. In CIDE, we used this principle of abstraction for removing annotated code safely.

To summarize, although there are approaches to address syntax, typing, and semantic errors in SPLs for single languages, there are no *language-independent* solutions. In the remainder of this paper, we explore language-independent checks for SPLs implemented with annotated source code. We begin by describing CIDE's guarantee for syntactic correctness in Java and subsequently generalize it for other languages.

3 Checking Syntactic Correctness of Java SPLs

In this section, we revisit CIDE, a tool for developing Java SPLs using annotations on source code, which we presented in prior work [22]. With simple design principles, CIDE guarantees that all variants are syntactically correct Java programs. To provide context, we first give an overview of the origins of CIDE and the motivation behind it.

CIDE was originally designed to analyze and discuss how code fragments that implement a feature are scattered and interact inside legacy applications. In our discussions, we originally highlighted those code fragments on printouts with text markers using a different color for each feature. This turned out to be useful, so that the motivation for CIDE was to convey this color metaphor to a Java IDE based on Eclipse. In CIDE, developers assign code fragments to features. These annotations are then represented with a background color in the editor (one color per feature), just as with the text marker on paper.[2] When creating a variant, code annotated with unwanted features is removed like when using *'#ifdef'* directives in C. The main visual difference lies in the fact that CIDE uses background colors instead of *'#ifdef'* directives for annotations.

Besides the visual representation, another key difference between CIDE and traditional preprocessors (a difference we later use to guarantee syntactic correctness) is that, in CIDE, features are assigned to elements of the underlying structure of the Java code, instead of assigning them to a sequence of characters (possibly determined by offset and length). As underlying structure, we use the *abstract syntax tree (AST)* that represents a Java artifact at a fine granularity. The AST provides flexibility to annotate even small code fragments in the middle of a method, which we often needed in our projects. In CIDE, we assign features to AST nodes that represent the selected code fragment, other annotations are not possible. Nevertheless, for users, the underlying structure is transparent, they simply annotate code fragments, which are then mapped to AST nodes internally.

In Figure 3, we illustrate this concept of using the underlying structure with a simple example. It shows a code fragment and its AST. When assigning a feature to a code

[2] In case multiple features are assigned to the code fragment, the corresponding background colors are blended. Though this does not allow to recognize feature code solely from the background colors, it indicates where feature code starts and ends, so that the user can lookup the actual features in a tool tip or even infer them from the context.

```
1 class C {
2   void m(int p){
3     s1();
4     s2(p,true);
5   }
6 }
```

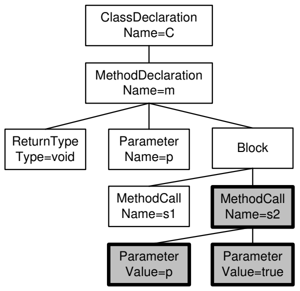

Fig. 3. Source code fragment and according AST.

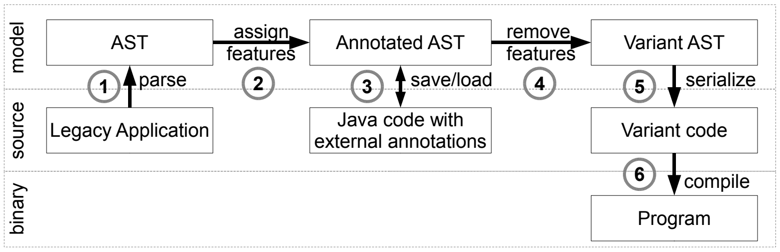

Fig. 4. CIDE's process for implementing an SPL and creating a variant

fragment (in this example, Line 4, underlined), the code fragment is internally mapped to the corresponding AST nodes (grayed). Code fragments that cannot be mapped to AST nodes cannot be annotated. In the user front-end, the code fragments belonging to annotated AST nodes are shown with a background color according to their assigned features (of course other visual representations would be possible as well, e.g., using additional keywords like *#ifdef* instead of colors).

The overall process to develop an SPL with CIDE is depicted in Figure 4. Developers begin with a syntactically correct base implementation (possibly a legacy application), which is parsed into an AST (Step 1). Then, they assign features to AST nodes inside the development environment (Step 2). The feature-annotated AST can be saved and loaded again (Step 3), so that it is still possible to edit the source code. To generate a variant, developers select a set of features and CIDE removes all AST nodes that are annotated with features that are not selected (Step 4). Note, this step is a transformation from one AST to another. The generated AST is then serialized ('unparsed') as source code (Step 5), which finally can be compiled into a program (Step 6).

Enforcing Syntactic Correctness. Representing source code as ASTs instead of plain text and the AST transformation in Step 4 are the key to CIDE's guarantee for syntactic correctness. With only two rules, we can ensure that every transformation in Step 4 transforms the annotated AST into another AST that also adheres to Java's syntax specification. Thus, we can prevent by construction the generation of code with incorrect syntax. The rules are:

- **Optional-Only Rule:** Only AST nodes that are *optional* according to the Java syntax specification (as described in [14]) can be removed. For example, we cannot remove a class's name without invalidating the AST, but we can remove a method. Incidentally, the AST provided by the Eclipse Java framework already enforces this rule; it only removes optional elements and throws exceptions otherwise.
- **Subtree Rule:** When an AST node is removed, all its child nodes must be removed as well. For example, when method *m* is removed in Figure 3 also its parameter and statements must be removed; when class *C* is removed all content therein must be removed as well. For CIDE that means, when a user annotates an AST node, CIDE automatically propagates this annotation to all subnodes.

In CIDE, these two rules provide a foundational mechanism for syntactic correctness. Even without consulting a feature model, this mechanism guarantees that no transformation can invalidate the AST. At the same time, it provides a fine granularity so that even individual statements or parameters can be annotated, as discussed in [22]. Nevertheless, we found some situations in Java in which these rules are too restrictive and more flexibility is needed. For those situations, we manually implemented an exception.

Due to the Subtree Rule, we were not able to independently annotate code fragments that wrap other code. A typical example is a *try-catch* statement as in Figure 5, which could belong to an exception handling feature. The *try-catch* statement wraps a code block "*s1();*". When deselecting the exception handling feature in a variant, the Subtree Rule would automatically remove all child elements including the wrapped code block. The same effect occurs with several other Java statements like *try-finally*, *synchronize*, *if*, *for*, *while*, and *do*, which wrap other statements and which we therefore call *wrapping elements*. To offer more flexibility, we allow an *exception to the Subtree Rule* in Java: When a user annotates a wrapping element, specific child elements can be excluded despite the Subtree Rule. When removing this wrapper in a variant, it is *replaced* with the wrapped element.

```
class C {
  void m(){
    try {
      s1();
    } catch(Exception e) {
      handleException(e);
    }
  }
}
```

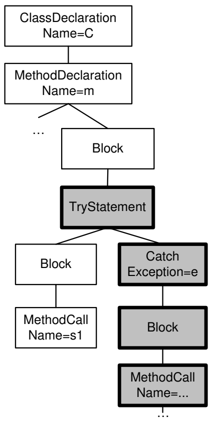

Fig. 5. Wrapper for try-catch Statement as Exception to the *Subtree Rule*.

4 Generalizing CIDE beyond Java

CIDE was created for Java and builds directly on Eclipse's Java framework, which made implementing the two rules simple because Eclipse already encodes the Java language specification. Nonetheless, SPLs usually consist of code and non-code artifacts written in different languages, e.g., source code, scripts, make files, documentation, models, or grammar files. All these artifacts should be handled uniformly by product line tools, as stated by the *principle of uniformity* [3, 1]. Therefore, our goal is to provide CIDE – including its guarantee for syntactic correctness – for multiple languages of different kinds of code and non-code artifacts.

In this section, we generalize CIDE beyond Java and proceed in two steps. First, we analyze the underlying principles behind the rules and exceptions we found for Java in order to derive a general model. Second, we describe our approach to automate the process of extending CIDE for new languages to minimize human effort.

4.1 Generalizing Correctness Rules: The gCIDE Model

CIDE's key mechanism for Java was abstracting from plain text, using the underlying structure, and allowing only operations on that structure which do not invalidate Java's syntax. To generalize these concepts, we need an underlying structure for other languages as well. This structure can be an AST as common in programming languages, a document object model as in XML, or another structure that represents the artifact.

In order not to implement distinct mechanisms for every language, we develop a *generalized model for CIDE (gCIDE model)* that language-independently represents a common underlying structure on which rules for syntactic correctness are defined. Thus, instead of statements, classes, AST nodes, XML elements, or others, we just generally speak of *structural elements*. For concrete languages, the language structure is mapped to this language-independent model. The full gCIDE model (which we explain next in several steps) is depicted in Figure 6.

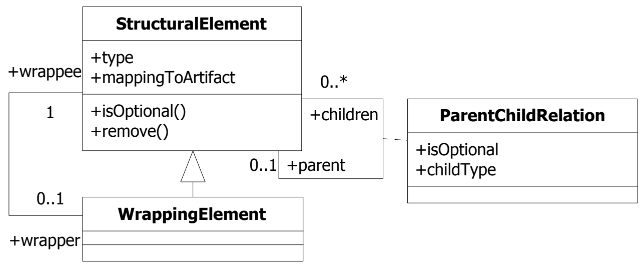

Fig. 6. gCIDE Model

Basics. The Subtree Rule can be applied directly to arbitrary tree structures: whenever a structural element is annotated, its children are annotated as well. In the gCIDE model, this tree structure is represented such that structural elements have exactly one parent each (except for the root that represents the entire file) and may have child elements which are again structural elements. For technical reasons, to be able to make the mapping between code fragments and structural elements transparent in CIDE, each structural element must store a mapping to the actual location in the artifact (modeled as *mappingToArtifact*).

To transfer the Optional-Only Rule from Java to other artifacts, there must be a description which elements in the tree structure are optional. In Java this can be derived from the Java Language Specification [14], in XML the allowed structure is specified by a W3C recommendation [5], for other languages such specification either exists or must be formulated to determine which (removal) transformations on the structure are safe. In the gCIDE model, independent of any specific language, the *isOptional* attribute of an element's relationship to its parent specifies whether the element can be removed safely. The values of the *isOptional* attribute must be assigned individually to every element for each language.

Wrappers and Types. For the wrapper exception to the Subtree Rule (removing a *try-catch* statement in Java without removing the inner statements), we introduce the notion of a wrapping element in our model (cf. Fig. 6). In the gCIDE model, a wrapping element is a special case of a structural element and specifies exactly one child element it wraps (implications of allowing to wrap multiple child elements are discussed below). When removed, it is replaced by this child element.

Wrapping elements cannot be placed at arbitrary places or wrap arbitrary elements. For example, a Java class cannot wrap a method such that the class is replaced by this method if removed, as this would invalidate the AST. In the original implementation of CIDE, we manually defined specific exceptions for selected Java elements and implemented them individually. For a general solution in gCIDE, we use a *type-based mechanism* instead.[3] Each structural element belongs to a type and there is a subtype relation on those types. Parent-child relations between structural types state the type of elements they expect. For example, Java classes expect members as child elements, blocks expect statements. The subtype relation describes which structural types represent a member or a statement and can be used as child element. For example, 'method' and 'field' are subtypes of 'member' and can be child elements of classes, while 'method invocation' and 'try-catch' are subtypes of 'statement' and can be used as a child element for blocks. Types, subtype relations, and accepted types are modeled in the gCIDE model and must be provided for each language.

With types, a straightforward algorithm can determine where wrappers are allowed. For this, it needs to consider three elements: the wrapper, its parent, and the wrapped child. The wrapped child is allowed if it is accepted as a child of the parent. For example, in Figure 5 the try-catch statement can wrap a block but not the *CatchBlock* element, because only the block is acceptable as child to the parent.[4]

[3] For clarification: the types used in this mechanism represent syntactical categories of the host language as statements, parameters, or blocks. They are not to be confused with types of terms in the host language.

[4] Note, it is conceptually possible to model structural elements that wrap multiple elements under some conditions, but it would require a more complex model and reasoning. Required conditions are: (1) all wrapped elements must be of the correct type and (2) it must be allowed to replace

4.2 Automating Language Plug-in Creation

Using the gCIDE model, we can now extend CIDE to support multiple languages. We evolved CIDE and removed all Java specific code and replaced it by an abstract framework following the gCIDE model. Concrete target languages can now be added as extensions – so called *language plug-ins* – which implement this framework. Thus, a language plug-in creates structural elements for a specific target language and fills values like types and *isOptional*. This way, we can separate the infrastructure that is common for all languages (user frontend, feature management, tree transformation, cf. Step 2 and 4 in Figure 4) from the implementation of specific target languages.

However, the effort to create language plug-ins for CIDE is still high. First, we need a parser for each language that transforms the artifact into the tree structure (Step 1 in Figure 4). Second, we need to serialize ('unparse') transformed structures in each language to write them back to an artifact (Step 5 in Figure 4). Finally, and most importantly, we need to define the rules and exceptions for each specific language: we need to define which structural elements are optional or wrappers, so that transformations always transform valid trees into other valid trees.

A straightforward way to develop language plug-ins is to bridge the internal structures of an existing open compiler or graphical editor to the gCIDE model. For example, we could bridge the AST from Eclipse's Java framework or the internal structure of an UML editor and thus reuse this infrastructure. However, for many languages, industrial-strength compilers that can be accessed and reused are not available. Furthermore, implementing the bridge might still require considerable effort. Instead, we pursue an approach, in which we can uniformly *generate* language plug-ins.[5]

Fortunately, creating language plug-ins (parser generation, serializer implementation, rule definition) can be automated to a high degree from the grammar of the target language, as we will show in the remainder of this section. First, existing parser generators can generate a parser from a grammar. Second, some parser generators can also create a 'pretty printer' that can be used for serialization ('unparsing'). Finally, even information for the Subtree Rule and the Optional-Only Rule can be derived from the target language's grammar, because a grammar specifies (1) the child-parent relationship between structural elements, (2) which elements are optional, and (3) subtyping information. That is, we can use the grammar of a target language as the single source to generate a language plug-in.

To generate language plug-ins for CIDE from grammar specifications, we built our own tool chain, because common parser generators (e.g., JavaCC, yacc, or ANTLR) do not propagate sufficient information from the grammar to the created tree structure. For example, from the parse tree that JavaCC generates, we cannot determine which elements are optional. Therefore, we defined our own grammar specification language called

one wrapper with multiple elements. Alternatively, more complex custom transformations could be specified. To keep the model simple, we disallow wrappers around a multiple elements.

[5] Note, our generation approach targets artifacts which are usually edited with a textual editor like source code or textual modeling and specification languages such as Alloy. For artifacts that are usually edited in a graphical editor such as UML, often a tailored approach of mapping the gCIDE model to the representation of a specific editor is more suitable.

FeatureBNF and built a tool called *astgen* which generates *LL(k)* parsers, serializers, and trees with all information required by the gCIDE model.[6]

FeatureBNF Basics. FeatureBNF uses an extended Backus-Naur notation and supports some additional annotations for *astgen*. From a given grammar, elements are recognized as optional when followed by a question mark, or when they are part of a list (expressed with an asterisk symbol).

In Figure 7, we illustrate an excerpt from a sample programming language. A compilation unit consists of any number of type declarations. The type declaration consists of one mandatory identifier, a second optional one, an optional 'implements' list, and a class body. The class body contains any number of fields or methods.

```
1 CompilationUnit : (TypeDeclaration)* <EOF> ;
2 TypeDeclaration : "class" <ID> ( "extends" <ID> )? ( ImplementsList )? ClassBody;
3 ClassBody       : "{" (Member)* "}" ;
4 Member          : Method | Field ;
5 ImplementsList  : "implements" <ID> ("," <ID>)* ;
```

Fig. 7. FeatureBNF grammar example

From a given grammar for a target language, *astgen* generates the language plug-in, consisting of a parser that builds a tree structure for a given artifact in that language, and a pretty printer that writes it back into a file. The tree structure generated by the parser not only represents the structure of the source code, but can also reflect its structural properties derived from the grammar, e.g., each tree node knows whether it is optional as described in the gCIDE model. So, in the given example in Figure 7, the type declarations are optional (there can be any number of type declarations in a compilation unit, Line 1) and thus can be annotated in CIDE. Inside the type declaration the first identifier and the class body are mandatory and cannot be annotated, but the second identifier and the 'implements' list are optional (Line 2). Also members are optional and can be annotated (Line 3). In a full grammar, typically also statements, parameters, or parts of expressions are optional and can be annotated in CIDE.

Concrete Syntax vs. Abstract Syntax. A parser generated from a grammar for a target language with standard parser generator tools creates a *Concrete Syntax Tree (CST)*. This tree contains those elements that are required for parsing. However, a CST does not necessarily reflect the abstract syntax of the language, and mapping a CST to the gCIDE model (instead of an AST) can result in reduced flexibility.

A typical example how the concrete syntax may reduce flexibility is the use of lists, as exemplified in Figure 7, Line 5. The 'implements' list is optional inside the

[6] Technically, the FeatureBNF grammar specification language is a meta-grammar. It is a grammar that specifies how developers can specify and annotate grammars for a specific target language. For example, we can write a Java grammar in the FeatureBNF format. From this Java grammar, *astgen* generates all required parts for a Java language plug-in. For parser generation, we internally reuse *JavaCC*. Note, FeatureBNF is reused with some extensions in another line of research on language-independent software composition [1].

type declaration, but inside the list the first entry is mandatory due to special parsing requirements for the separating comma. Using the CST, the first entry cannot be annotated individually, although in the abstract syntax all entries are optional elements of a list.

To ensure the full flexibility of the abstract syntax, it is necessary to transform the CST into an AST. In many tools this is a separate step after parsing. For serialization, the inverse transformation must be performed on the modified AST. To guarantee syntactic correctness for all variants, both transformations must be performed safely without loss of information.

To bridge this gap, we follow the lead of Wile, who used an extended grammar specification language to derive the abstract syntax directly from a grammar file [37]. Wile proposed a series of additional constructs in the grammar specification language, so that the abstract syntax and its relationship to the concrete syntax are directly specified in the extended grammar file. This way, we can generate a parser that directly produces an AST instead of an CST. Wile further proposed a semi-automated process to transform an existing grammar describing a concrete syntax into the extended format. For example, to solve problems like the 'implements' list described above, he proposes a special 'list' construct. In Wile's notation, the *ImplementsList* production is expressed as `ImplementsList: ID ^ ",";`, in which the ^ symbol is a special construct for lists followed by the token that separates list entries. Using this construct, the parser can interpret identifiers directly as lists and build the AST accordingly. We adopted Wile's concept and added those extensions that are relevant for our case studies in FeatureBNF. This way, we can generate a parser that creates structural elements based on the target language's abstract syntax from a grammar file. Only a single tree is created, no manual mapping between CST and AST is required, and all further transformations to remove annotated fragments can be directly performed on the AST of the artifact.

Wrappers and Other Exceptions. Although technically possible, wrappers are not automatically recognized from the grammar for a target language, but a language expert has to decide where wrappers make sense. Wrappers should only be used where the additional flexibility is needed. Otherwise, it would still enforce syntactic correctness, but be harder to use.[7]

FeatureBNF supports additional constructs to specify exceptions like wrappers. Other exceptions, e.g., making a mandatory production optional by providing a default value, or marking an optional production mandatory, can also be defined safely. Due to space restrictions we defer the interested reader to the language description at CIDE's web site.

5 Experience

After extending CIDE and building the tool infrastructure, we generated 15 language plug-ins from grammars of various code and non-code languages. Next, we conducted

[7] For example, classes could automatically be interpreted as wrappers around inner classes in Java. While this is syntactically correct and also fulfills the typing rules, annotating a whole class except an inner class, usually does not make sense. Offering such flexibility is only confusing to the developer; workarounds by rewriting the code are much easier to use.

Language	Optional structures	Wrappers	#Prod.
Featherweight Java*	methods, fields, parameters	-	16
Java 1.5**	members, statements, parameters, ...	if, for, try, ...	133
C (plain)**	functions, statements, parameters, ...	if, for, ...	80
C (pseudo)*	functions, statements, preprocessor, ...	if, #ifdef, ...	46
C++ (pseudo)*	classes, methods, statements, prepr., ...	if, #ifdef, ...	65
C#***	classes, members, statements, parameters, ...	if, for, try, ...	215
Haskell (pseudo)**	types, imports, data, classes, ...	-	54
Haskell 98*	types, imports, data, classes, parameters, ...	if	71
JavaScript/ECMAScript**	functions, statements, expressions, ...	if, for, ...	111
JavaCC**, Bali**, ANTLR***	productions, terminals, ...	[], ()*	14–166
Property files*	lines	-	1
HTML**	headings, paragraphs, list items, ...	<b></b>, ...	11
XML*	nodes, parameters	-	13

*handwritten based on external specification, **adapted from JavaCC grammar, ***adapted from ANTLR grammar

Table 1. CIDE Language plug-ins generated from FeatureBNF grammar

several small case studies applying CIDE to different projects to test its practicality regarding those languages. Due to space restrictions, we only give an overview of language plug-ins and SPLs projects. Further information is provided in an accompanying technical report [23] and on CIDE's website which contains the source code of all language plug-ins and SPL projects (except for the water boiler SPL which we cannot disclose to protect intellectual properties of our partners).

Language plug-ins. In Table 1, we list all supported languages and some information describing optional structures, wrappers, and the number of production rules each. The number of production rules can be used as a rough indicator of the complexity of the language. For most languages the FeatureBNF grammar was derived from an existing grammar in the JavaCC or ANTLR format (which required mostly just syntactic changes) and usually took less than one hour.

The language extensions for C and C++ were the most problematic, due to C's preprocessor. Although we could straightforwardly adapt an existing JavaCC grammar for C, this would only work on C code that was already preprocessed. That is, a C parser requires source code in which *'#include'* statements were resolved, *'#ifdef'* statements were removed, and macros were evaluated. Without preprocessing (or even just with ignoring preprocessor directives) C code cannot be parsed. Unfortunately, working on preprocessed code is only an option for downstream tools, but in CIDE developers work on unprocessed code that still contains preprocessor directives. This problem was already faced much earlier, e.g., by intentional programming [31] or refactoring tools [13] and required complex workarounds. To overcome this problem in CIDE, we wrote a pseudo-parser, which does not actually parse the code based on the full language specification, but recognizes only important constructs like functions, variable declarations, or statements. For example, statements are recognized by the terminating semicolon, functions by the typical pattern for return type and parameter declarations. Preprocessor directives are

Project	Features	LOC	Annotated Parts	Time
Berkeley DB	38	204 000	Code (Java), Documentation (HTML)	4 days
Graph Product Line	14	2 300	Code (Java), Documentation (XHTML)	3 hours
AHEAD Tool Suite	24	45 000	Doc. (HTML), Build scripts (XML)	2 hours
SQL Parser SPL*	4	60	Grammar (ANTLR)	20 min
Arithmetic SPL*	3	120	Code (Haskell)	1 hour
FAME-DBMS*	14	6 000	Code (C++)	2 days
Water Boiler SPL**	14	10 000	Code (C)	2 days

*prototype, **closed-source, industrial

Table 2. SPL projects implemented with CIDE

recognized as part of the language, as long as they are used within certain limitations (e.g. *'#include'* may not occur inside functions). With this pseudo-parser, we are able to use CIDE on C projects, but we weaken the guarantee for syntactic correctness to some degree (see discussion in Sec. 6). The same pseudo-parser approach was used for C++, and also for an initial version of Haskell, because of Haskell's complex syntax.

The XML grammar supports only plain XML files, i.e., all elements or parameters are optional and can be annotated. This guarantees that every variant is *well-formed* in the XML terminology. Guaranteeing that all variants of an XML artifact are *valid* based on the given DTD or XML Schema description requires additional information. Note, DTD is a meta-grammar itself that can be used for XML documents instead of FeatureBNF. We implemented a prototype transformation tool *dtdgen*, which converts a DTD into a FeatureBNF grammar, as a proof of concept. This way, we generated a parser specifically for XHTML (version '1.0 strict'). However, in ongoing work, we pursue a direct transformation from a DTD or XML Schema to the gCIDE model by extending an off-the-shelf XML parser [11].

Finally, for the Java language plug-in it is worth emphasizing that it is also generated from a grammar and not bridged from the Eclipse Java framework. Still, for enforcing syntactic correctness, the generated version is as flexible as the original one.

All in all our experience shows that creating language plug-ins for new languages is simple for a variety of different languages, even for complex or less popular languages like Haskell or JavaScript. If a target language has a well-specified grammar, creating a language extension is a matter of few hours. All generated languages share CIDE's guarantee for syntactical correctness (with some limitations for pseudo-parsers).

SPL projects. In order to demonstrate that CIDE can actually be used for developing non-trivial SPLs written in different languages, we used CIDE in a series of small to medium-sized projects, as listed in Table 2. For brevity, we again only give an overview and refer the interested reader to our accompanying technical report [23]. The given durations for each project are rough estimates, and it has to be taken into account that in most projects the features were previously known, so only little code exploration was necessary. However, the point of these projects is not to discuss the feasibility or effort of creating SPLs by annotating legacy applications, but we focus on CIDE's applicability to different languages.

The first two projects are interesting, because with the generalized version, we could not only annotate the Java code of these SPLs, but also the documentation (e.g., 120 000 lines of HTML in Berkeley DB). This way, the documentation is also tailored specific for the generated variant, e.g., for a Berkeley DB variant without transactions, the ‘Transaction Processing Guide’ and all references to it are removed. In the third project, we additionally annotated the build scripts which are ANT files in XML format. Despite their small size, the prototypes of SQL Parser SPL for embedded systems and Arithmetic SPL demonstrate CIDE’s capabilities to languages beyond mainstream object-oriented languages. Finally, the last two projects are again real projects for embedded systems written in C and C++ from academia and industry that are actively developed and maintained. Even though, we provide only a pseudo-parser for C and C++, we could successfully annotate these projects. In these projects, we experienced tedious syntax errors quite frequently in earlier implementations before using CIDE.

In all projects, we were able to create different variants and found (using tools, tests, or manual evaluation) that all generated variants are syntactically correct. For example, for FAME-DBMS and the Water Boiler SPL we successfully generated, compiled, and executed different variants.

6 Discussion: Flexibility vs. Safety

During development and testing, we found a trade off between two properties: flexibility and safety. By imposing a tree structure on a source code artifact, we provide *safety* (guaranteeing syntactic correctness for all generated variants). However, at the same time, we impose restrictions on what developers are allowed to annotate and thus reduce their *flexibility*, i.e., they have fewer possibilities to express variability. For example, compared to an implementation using the C preprocessor that works on token level[8], CIDE only annotates optional elements of the underlying AST. Hence, using such preprocessor, a method can have two alternative return types, but in CIDE a return type cannot be annotated independently when it is mandatory in the language’s grammar.

Programming languages already define a certain structure for code artifacts by their language syntax. For example, the Java syntax defines a top-down structure for Java code (e.g., Java files contain classes, which contain methods, which contain statements). By using ASTs to annotate programs, we expose this structure in CIDE and employ it with the Subtree Rule and the Optional-Only Rule to prevent syntax errors. Different languages provide a different amount of structure in their syntax. For example, JavaScript artifacts only contain a list of statements or function declarations, grammar artifacts only contain a list of production rules with a simple inner structure, and XML nodes are nested completely arbitrarily.

This raises two questions. (1) How much structure does an artifact language need to be usable in CIDE? (2) Is the structure defined by a language’s syntax a limitation when adding further language plug-ins for other artifact types in the future?

[8] To be precise the C preprocessor works on lines of source code. However, due to the code layout flexibility in many languages, it can usually be used for token-based annotations.

To answer the second question first, consider a ‘README.txt’ file. It is a valid artifact in an SPL, but will probably not provide any structure, at least none that is described by a *LL(k)* grammar. Fortunately, such artifacts, for which no specific language grammar is specified, can still be parsed by a dummy grammar that matches any file as a list of arbitrary optional tokens or even as a list of optional characters. With a dummy grammar, every character is optional with respect to the document, i.e., every single character in this document can be annotated independently just as when using preprocessors. This shows that a required structure is not a limitation of our approach. Even if no structure is available, as in the ‘README.txt’ file, we can still use the same tool to annotate this file uniformly next to other artifacts in an SPL.

Nevertheless, structure is beneficial. When using the dummy grammar, the guarantee of syntactic correctness is lost, because *any* artifact adheres to this grammar. This shows that any structure – although it reduces flexibility – is beneficial for safety, because the grammar defines CIDE’s syntax checks. It restricts the possible parts of the artifact that can be annotated and enforces ‘reasonable’ annotations.

In this context, the pseudo-parsing approach presented for C, C++, and Haskell artifacts in Section 5 is interesting. Pseudo-parsing does not use the full structure as provided by the language grammar (in these particular cases because of technical limitations caused by the preprocessor in C/C++ and because of Haskell’s complexity). Instead, it uses a simpler approach that recognizes only certain elements like functions and statements, but ignores inner fragments like parameters, or expressions. There are two possibilities to handle those inner fragments which are ignored by the parser. First, we can regard these fragments as a single mandatory node each, i.e., there is no substructure inside such fragment (used in Haskell, referred to as alternative A below). Alternatively, we can parse these fragments with a dummy grammar as a list of optional tokens or characters (used in C and C++, referred to as alternative B). Pseudo-parsing can be used to balance between flexibility and safety, as alternative A guarantees syntactic correctness at a significantly reduced flexibility, while alternative B is more flexible but guarantees syntactic correctness only for the recognized parts and not for their inner structure.

In Figure 8, we visualize the relative differences in safety and flexibility of all approaches. For annotations based on a concrete syntax tree or an abstract syntax tree, we can guarantee syntactic correctness, while the AST provides more flexibility (see Sec. 4.2). Pseudo-parsing approaches in which the internal structure of recognized elements is opaque (alternative A) can also guarantee safety, but with a significantly reduced flexibility. In contrast, a character based or token based annotation – as with the dummy grammar or *‘#ifdef’* preprocessors – provides the highest flexibility (every single character or token can be annotated) but no safety at all. A pseudo-parser that uses a dummy grammar for inner elements (alternative B) lies in the middle, it provides high flexibility, but reduced safety.

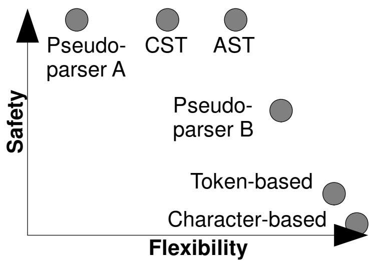

Fig. 8. Safety vs. Flexibility

This discussion shows that a structure given by a grammar is not necessary for an artifact to be handled by CIDE. However, as we experienced in our case studies, when a reasonable grammar is provided, CIDE can ensure syntactic correctness and take advantage of the artifact's structure to support the developer toward reasonable annotations. The ability to use the artifact's structure (if available) in every language to ensure syntactic correctness distinguishes CIDE from naive *'#ifdef'*-like preprocessor approaches.

7 Perspective: Language-Independent Checks Beyond Syntax

We showed how CIDE guarantees that every variant of an SPL is syntactically correct, independent of the artifact's language. This is helpful to prevent errors when developing SPLs, and in fact it this is more than provided by any current language-independent SPL technologies and tools like Frames/XVCL [18], C preprocessor, *pure::variants* [4], or *Gears* [26]. Still, many other kinds of errors are possible in CIDE (see Taxonomy in Sec. 2). In the remainder, we provide an outlook on future work to detect also typing and semantic errors, while still supporting multiple languages. We addressed syntactic correctness in this paper first, because it is a necessary precondition for all these additional checks.

Detecting Type Errors. CIDE, as described so far, covers only the syntax of a language, not its type system. In a variant that is syntactically correct, there can still be type errors like dangling method invocations (when only the method declaration but not the invocation has been removed in this variant). Type errors can be detected in many (statically typed) languages by static program analysis. Ideally, an SPL tool should be able to check both syntax and typing in multiple languages in order to detect all compilation errors without actually compiling all variants.

Based on CIDE, we implemented type checks for SPLs written in Java that can guarantee that all variants are well-typed once certain checks pass [21, 25, 23]. If there is a problem like a dangling method reference – even though it might only occur in few variants – an error is reported in CIDE. For Featherweight Java (a subset of Java), we have formalized these type-checks and proved them complete [21].

The basic mechanism of these type-checks is simple and can be generalized to other languages. Type checks are broken down into pairs of code elements that reference each other. For example, a method invocation references a method declaration; a field access references a field declaration; in UML an association references two elements; and so on. For these pairs, CIDE makes sure that in every possible variant in that the referencing element (e.g., method invocation) is present, also the referenced element (e.g., method declaration) is reachable. This condition can be expressed based on the elements' annotations and the SPL's feature model and evaluated using a SAT solver as described in [21].

The agenda for providing type checks for multiple languages is similar to the path we took for syntax checks. As first step, we generalized the common parts of type checking SPLs in a framework in CIDE, similar to the gCIDE model. We successively use this framework to implement type-checks for language plug-ins, currently Java and Bali [3] are implemented. Whether these steps can be automated is an open research question.

Detecting Inter-Language Type Errors. Another interesting question, when dealing with multiple languages inside an SPL is inter-language typing. It is often not sufficient to check artifacts from a single language, but annotated artifacts from different languages may reference each other and must be consistent in all variants – e.g., a web service description (WSDL file) in XML format references the implementation in a programming language like C#.

As type checks in a single language, checks can be broken down to pairs of elements that reference each other. However, language plug-ins must be able to detect these inter-language pairs. The main research questions are finding the right abstractions and a suitable polylingual type system (e.g., [15]) for these inter-language checks of SPLs. Advances in inter-language refactorings in Eclipse can be used as starting point [12].

Detecting Semantic Errors. Semantic errors are difficult to detect even without SPL technologies. While there are several approaches of how to formally specify behavior of programs and how to automatically check these specifications, they hardly scale for industrial-size programs and are used only in few scenarios. First steps for verifying SPL behavior against some formal specifications [36] and for using annotations in an SPL for model checking [30] were suggested.

To support such verification and model-checking mechanisms in CIDE, we yet again have to find the right abstractions and extend language plug-in mechanism to map the according language-specific conditions to a generalized model for CIDE. Still, there are open research questions how to provide or automate such checks in CIDE and how to scale them for large SPLs. Nevertheless, this is a promising avenue for future work, especially for safety critical SPLs in embedded systems.

8 Conclusion

Software product lines (SPLs) usually contain artifacts written in different languages. To handle different artifacts uniformly, current SPL technologies either (a) use an approach that is so general that it works for arbitrary artifacts, but can easily introduce subtle errors for some variants (preprocessors, XVCL, Gears, pure::variants, etc.); or (b) they provide specialized tools for a low number of languages (frameworks, AHEAD, aspects, generators, etc.). Errors that only occur in certain variants of the SPL are a serious problem, as the exploding number of variants makes testing by generating, compiling, and running each variant infeasible.

We have shown how CIDE, an SPL development tool that can be considered as disciplined preprocessor, guarantees syntactic correctness for all variants that the SPL can generate by abstracting from the concrete textual representation in a file and using its internal structure. We have shown the underlying principles and generalized them from Java into a language-independent model. In a further step, we have even automated the process of creating language plug-ins from annotated grammar files, so that extending CIDE (including its guarantee for syntactical correctness) for new languages requires minimal human effort.

We have demonstrated CIDE's applicability by generating plug-ins for a series of code and non-code languages including Java, C, C#, Haskell, JavaScript, ANTLR, and

XML. We have further shown CIDE's usability for concrete problems in several different case studies that consist of artifacts written in different languages.

Acknowledgments. We thank Peter Kim for fruitful comments on earlier drafts of this paper, Marko Rosenmüller and Norbert Siegmund for their help with C and C++, Armin Größlinger for help with Haskell and for providing programs for the Haskell case study, and Sagar Sunkle for releasing the annotated SQL grammars. Apel's work was funded partly by the German Research Foundation, project #AP 206/2-1. Trujillo's work was co-supported by the Spanish Ministry of Science & Education under contract TIN2008-06507-C02-02. Batory's work was supported by NSF's Science of Design Project #CCF-0724979.

References

1. S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automatic Software Composition. In *Proc. Int'l Conf. on Software Engineering (ICSE)*. 2009.
2. S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.
3. D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, 2004.
4. D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comput. Program.*, 53(3):333–352, 2004.
5. T. Bray et al. Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation, W3C, 2006.
6. P. Clements and C. Krueger. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *IEEE Software*, 19(4):28–31, 2002.
7. P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
8. K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press, 2000.
9. K. Czarnecki and K. Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proc. Int'l Conf. Generative Programming and Component Eng. (GPCE)*, pages 211–220. 2006.
10. B. Delaware, W. Cook, and D. Batory. A Machine-Checked Model of Safe Composition. In *Proc. AOSD Workshop on Foundations of Aspect-Oriented Languages (FOAL)*, pages 31–35. 2009.
11. J. Dörre. Feature-Oriented Composition of XML Artifacts. Master's thesis, University of Passau, Germany, 2009.
12. R. Fuhrer, M. Keller, and A. Kieżun. Advanced Refactoring in the Eclipse JDT: Past, Present, and Future. In *Proc. ECOOP Workshop on Refactoring Tools (WRT)*, pages 31–32, 2007.
13. A. Garrido. *Program Refactoring in the Presence of Preprocessor Directives*. PhD thesis, University of Illinois at Urbana-Champaign, 2005.
14. J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM}Language Specification*. The Java$^{\text{TM}}$Series. Addison-Wesley Professional, 3 edition, 2005.
15. M. Grechanik, D. Batory, and D. Perry. Design of Large-Scale Polylingual Systems. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 357–366. 2004.
16. S. Huang, D. Zook, and Y. Smaragdakis. Statically Safe Program Generation with SafeGen. In *Proc. Int'l Conf. Generative Programming and Component Eng. (GPCE)*, pages 309–326. 2005.

17. S. S. Huang and Y. Smaragdakis. Expressive and Safe Static Reflection with MorphJ. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 79–89. 2008.
18. S. Jarzabek et al. XVCL: XML-based Variant Configuration Language. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 810–811. 2003.
19. R. E. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.
20. K. Kang et al. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990.
21. C. Kästner and S. Apel. Type-checking Software Product Lines - A Formal Approach. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 258–267. 2008.
22. C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 311–320. 2008.
23. C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-Independent Safe Decomposition of Legacy Applications into Features. Technical Report 2/08, School of Computer Science, University of Magdeburg, Germany, 2008.
24. G. Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. 1997.
25. C. H. P. Kim, C. Kästner, and D. Batory. On the Modularity of Feature Interactions. In *Proc. Int'l Conf. Generative Programming and Component Eng. (GPCE)*, pages 23–34. 2008.
26. C. Krueger. Easing the Transition to Software Mass Customization. In *Proc. Int'l Workshop on Software Product-Family Eng.*, pages 282–293. 2002.
27. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.
28. K. Pohl and A. Metzger. Software Product Line Testing. *Commun. ACM*, 49(12):78–81, 2006.
29. M. Poppleton, B. Fischer, C. Franklin, A. Gondal, C. Snook, and J. Sorge. Towards Reuse with "Feature-Oriented Event-B". In *Proc. GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering*, 2008.
30. H. Post and C. Sinz. Configuration Lifting: Verification meets Software Configuration. In *Proc. Int'l Conf. Automated Software Engineering (ASE)*, pages 347–350, 2008.
31. C. Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In *NATO Science Committee Conference*, 1995.
32. H. Spencer and G. Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, pages 185–198, 1992.
33. M. Staples and D. Hill. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *Proc. Asia-Pacific Software Engineering Conf. (APSEC)*, pages 176–183. 2004.
34. A. Tevanlinna, J. Taina, and R. Kauppinen. Product Family Testing: a Survey. *SIGSOFT Softw. Eng. Notes*, 29(2):12–12, 2004.
35. S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l Conf. Generative Programming and Component Eng. (GPCE)*, pages 95–104. 2007.
36. E. Uzuncaova, D. Garcia, S. Khurshid, and D. Batory. A Specification-Based Approach to Testing Software Product Lines. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 525–528. 2007.
37. D. Wile. Abstract Syntax from Concrete Syntax. In *Proc. Int'l Conf. on Software Engineering (ICSE)*, pages 472–480. 1997.