Technical Report 2/2008, School of Computer Science, University of Magdeburg, March 2008

Language-Independent Safe Decomposition of Legacy Applications into Features

Christian Kästner
School of Computer Science
University of Magdeburg
39106 Magdeburg, Germany
ckaestne@ovgu.de

Sven Apel
Dept. of Informatics and Math.
University of Passau
94030 Passau, Germany
apel@uni-passau.de

Salvador Trujillo
IKERLAN Research Centre
Mondragon, Spain
STrujillo@ikerlan.es

Martin Kuhlemann
School of Computer Science
University of Magdeburg
39106 Magdeburg, Germany
mkuhlema@ovgu.de

Don Batory
Dept. of Computer Sciences
University of Texas at Austin
Austin, Texas 78712, USA
batory@cs.utexas.edu

Abstract

Software product lines (SPL) *usually consist of code and non-code artifacts written in different languages. Often they are created by decomposing legacy applications into features. To handle different artifacts uniformly (code, documentation, models, etc.), current SPL technologies either use an approach that is so general that it works on character or token level, but can easily introduce subtle errors; or they provide specialized tools for a low number of languages. Syntax errors that only occur in certain variants are difficult to detect, as the exploding number of variants makes a manual testing unfeasible. In this paper, we present* CIDE*, an generic SPL tool that can ensure syntactic correctness for all variants. We show CIDE's underlying mechanism that abstracts from textual representation and* generalize *it from Java to arbitrary languages. Furthermore, we* automate *the generation of safe plug-ins for additional languages from annotated grammars. To demonstrate CIDE's capabilities, we applied it to a series of case studies with artifacts from different languages, including Java, C#, C, Haskell, ANTLR, and XML.*

1 Introduction

A *Software Product Line (SPL)* is a set of software-intensive systems that share a common, managed set of features, satisfying the specific needs of a domain [9]. Features are increments in program functionality and can be implemented with a wide range of mechanisms. By selecting a set of features, it is possible to generate a program – a member of the SPL, called *variant* – tailored to a specific application scenario. In practice, a common way to create an SPL is to refactor one or more existing applications in order to extract features, instead of designing the SPL from scratch, as this promises less risk and faster results [8].

In prior work, we presented the *Colored Integrated Development Environment (CIDE)* for building SPLs in Java with fine-grained extensions as they are especially needed when extracting features from legacy applications [27]. CIDE follows a paradigm we call *virtual separation of concerns*, i.e., developers do not physically extract the feature code, but just annotate code fragments inside the original code and use tool support for views and navigation. To generate a variant, CIDE removes code associated with those features that are not required. CIDE is similar to preprocessors as in C, however, the code is not polluted with *'#ifdef'* statements, but features are highlighted as background colors in the editor. If a physical separation of concerns is necessary, CIDE can export the colored code into feature modules (cohesive pieces of code that implement features) implemented in Jak or AspectJ [28]. A more extensive introduction to the concepts behind CIDE is given in Section 3.

When decomposing a legacy application into features, a typical challenge is to ensure that the decomposition is safe, i.e., that all variants are correct (syntactically correct, well-typed, correct behavior, etc.) [34, 12, 42, 44]. Though CIDE was originally designed to analyze and visualize feature interactions for Java [32], we found that the basic mechanisms we used allow us to claim that *every variant generated with CIDE is syntactically correct*. In this paper, we illustrate this mechanism and we use it as starting point to analyze its underlying principles for a generalization.

This led us to a second challenge. CIDE was originally

designed for Java artifacts only. However, as stated by the *principle of uniformity* [5], SPLs are typically built from a number of different code and non-code artifacts (e.g., source code, models, specifications, grammars, build scripts, documentation) that should be treated uniformly by SPL tools. Therefore, in this paper, we *generalize* CIDE to arbitrary languages while preserving the property that every variant is syntactically correct. Nevertheless, a formal generalization is not sufficient, as we still need to implement instances of CIDE for each language. Thus, in this paper, we also *automate* the process of extending CIDE for an additional language from the language's grammar.

The paper is structured as follows. First, we show the mechanism used to ensure syntactic correctness in Java. Second, we generalize these mechanisms to transfer them to arbitrary languages, so that CIDE can be used to generate program variants safely for any language. Third, we automate the creation of language plug-ins from the language's grammar, and show how we applied this approach successfully to a series of code and non-code languages. Finally, we give a perspective how CIDE can be extended to not only guarantee syntactically correct program variants, but also that variants are well-typed and even consistent in polylingual systems. The overall goal of this endeavor is to ensure a language-independent safe decomposition of complex systems that are typically written in multiple languages into features. Our latest version of CIDE together with all languages and case studies presented in this paper can be downloaded from the project's web site: `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/`.

2 Problem Statement

There are two main approaches to decompose a legacy application into features in order to create an SPL. The first approach is to physically extract features into new modules and compose a selection of them to create variants. There are several technologies that can be used for this *compositional approach*, including frameworks [24], components [41], AHEAD [5], aspects [29], or aspectual feature modules [2]. The composition mechanism usually assure that – if all modules are syntactically correct – the resulting variant is syntactically correct as well. However, these technologies are often limited to a single or a low number of different languages. Even though the AHEAD Tool Suite pioneered the approach of composing features written in multiple languages (e.g., Jak, bali, XML), still a tool for each language was developed individually [5]; only recent work addresses this issue [3]. Furthermore, although decomposition into features is possible with the compositional approach, the effort is comparatively high [32, 43, 36, 26], and only a low compositional granularity is supported, which may hamper the refactoring [27].

The *annotative approach* is an alternative, in that legacy code is not physically decomposed but just annotated; a variant is created by modifying or removing annotated code. Typical technologies include *'#ifdef'* preprocessor directives, Frames/XVCL [23], or tools like *Gears* [30]. These technologies are often so general that they are language-independent because they treat all artifacts as ordinary pieces of text. At the same time, they can introduce subtle errors that only occur in some variants when an application engineer actually creates them.

Consider the code fragment in Figure 1 that shows a fragment of C code from Berkeley DB[1] that uses C's preprocessor with multiple (partly nested) *'#ifdef'* directives to be able to generate different variants of the database engine. Already, this short code fragment illustrates the complexity that may occur in decomposition, 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 all other variants will result in a syntax error.

```
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
```

Figure 1. Broken code excerpt of Berkeley DB.

The shown problem is trivial, but when decomposing a legacy application by successively adding such annotations, syntax errors can easily occur in some variants. They can be difficult to detect, especially when they occur in nested annotations and thus only in very few variants [39]. A brute force strategy of generating and compiling all variants is usually not feasible, because already with few features the number of program variants that can be generated from an SPL explodes exponentially. For n independent optional features, there are 2^n distinct variants; an SPL with 320 features could generate more variants than there are estimated atoms in the universe. Therefore, an approach that conceptually ensures that *every variant* is syntactically correct is very helpful in a software product line tool to avoid errors early on.

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

Interestingly, there are some 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* [11] to generate variants of UML models. In this tool syntax errors cannot occur because generation of a variant is not performed on the source code representation of the model (e.g., XML file), but on an abstract level with the Rational Software Modeler engine. The engine does not allow transformations that would invalidate the UML syntax (e.g., by removing only the name of a class). In other fields of software engineering, this abstraction principle is also frequently applied. For example, automated refactorings [14] are not performed directly on the source code, but on an abstract representation, e.g., on the abstract syntax tree. In CIDE we used the principle of abstraction for removing annotated code. Although originally intended for another purpose, it guarantees syntactic correctness for Java code, as we will show in the next section.

Although there are tools that guarantee syntactic correctness for few specific languages, to the best of our knowledge, there is no tool that is *language-independent*. In the remainder of this paper, we will address this issue by generalizing CIDE to arbitrary languages.

3 Syntactic Correctness for Java

In this section, we introduce the basic design principles of CIDE and explain how they guarantee for Java that all variants of an SPL that can be generated are syntactically correct. For that purpose, it is first necessary to give an overview of the origins of CIDE and the motivation behind it.

CIDE was originally designed to analyze and discuss how the code fragments that implement a feature are scattered and interact inside legacy applications. In our discussions, we typically marked feature code with a text marker or used colored pens with a different color for each feature. This turned out to be very useful, so that the motivation for CIDE was to convey this color metaphor to a Java IDE based on Eclipse. In CIDE, developers assign features to code fragments, that are then highlighted with a background color in the editor (one color per feature), just as with the text marker on paper. We refer to this process simply as *'coloring code'*. Due to this metaphor, we avoid adding explicit annotations like *'#ifdef'* to the source code, but store feature annotations separately.

A key decision at the very beginning of the implementation that turned out valuable was to assign features to elements of the underlying structure of the Java file, instead of assigning it to a sequence of characters (possibly determined by offset and length). In order to achieve high flexibility (the discussed code fragments were often statement inside methods), we use the *abstract syntax tree (AST)* that represents

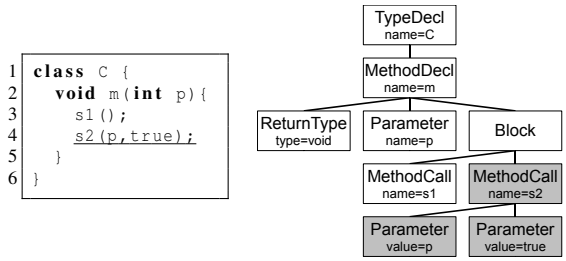

Figure 2. AST Example.

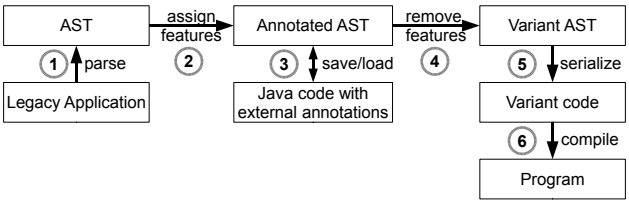

Figure 3. Decomposing a legacy application and creating a variant with CIDE.

the Java artifact and we assign colors to those AST nodes that represent the selected code fragment.

In Figure 2, we illustrate this concept on a simple example. It shows a code fragment and its AST. When assigning a feature to a code fragment (in this example Line 4, underlined) internally this is mapped to the corresponding AST nodes (gray boxes). In the user frontend, the code fragment belonging to an AST node is shown with a background color according to its assigned features. In case multiple features are assigned to the same AST node, the background colors are blended. Though this does not allow to recognize feature code solely from the background colors, it indicates where a feature's code starts and ends so that the user can lookup the actual features manually in a tool tip or context menu.

The overall process to get from a legacy application to an SPL in order to generate variants of the product is depicted in Figure 3. Developers start with 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 create a variant, developers select a set of features and CIDE removes all AST nodes that are annotated with features not required (Step 4). Note, this is a transformation from one AST to another. The generated AST is then serialized as source code (Step 5), which finally can be compiled into a program (Step 6).

Safe Decomposition Rules. Ensuring that every variant is syntactically correct relies on the mechanism of assigning

features to the underlying AST and to generate variants by AST transformations. With only two simple rules, we can ensure that every transformation in Step 4 transforms the annotated AST into another AST that also adheres to the Java syntax specification. Thus, we can prevent the generation of code with incorrect syntax in the first place. The rules are:

- **Optional-Only Rule:** Only AST nodes that are *optional* in the language syntax specification can be removed. For example, we cannot remove soley a class' 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 allows to remove optional elements only 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 *s2* is removed in Figure 2 also both parameters must be removed; when class *C* is removed all content therein must be removed as well. For CIDE that means, when an AST node is 'colored' all subnodes are 'colored' with the same feature as well.

In CIDE, these two rules provide a very simple mechanism to syntactic correctness. With them, no transformation is possible that invalidates the AST. At the same time, they provide a fine granularity so that even individual statements or parameters can be extracted, cf. [27]. Nevertheless, we found some situations in Java in which the rules are too restrictive and more flexibility is needed. For those situations we manually implemented specific exceptions.

Due to the Subtree Rule, we were not able to color code fragments that wrap around other code independently. A typical example is a *try-catch* statement, which belongs to an exception handling feature as shown in Figure 4. The *try-catch* statement wraps around a code block. When developers want to remove the exception handling (underlined) for some variants, then the Subtree Rule automatically removes all child elements including the wrapped code block (cf. AST in Figure 4). The same effect occurs with several other statements like *try-finally*, *synchronize*, *if*, *for*, *while*, and *do*, which wrap other statements and which we therefore call *wrapping elements*. To offer the flexibility needed to color wrapping elements, we allow an *exception to the Subtree Rule* in Java: When wrapping elements are colored, specific child elements can be uncolored despite the Subtree Rule. When removing the colored wrapper in a variant, it is *replaced* with the uncolored wrapped element.

4 Generalizing CIDE

CIDE was created for Java and builds directly on Eclipse's Java framework, which made implementing the two rules

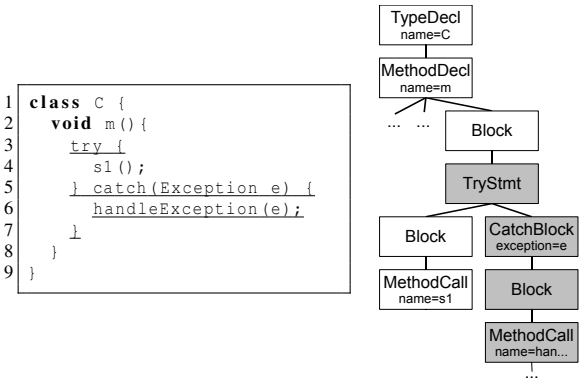

Figure 4. Wrapper as Exception to the *Subtree Rule*.

simple. Nonetheless, SPLs usually consist of code and non-code artifacts written in different languages, e.g., source code, build scripts, documentation, models, or grammar files. The *principle of uniformity* states that all these artifacts should be handled uniformly by the product line technology [5, 3]. Therefore, we want to offer CIDE for various languages, i.e., for decomposing different code and non-code artifacts, but still guarantee syntactic correctness. In this section, we generalize CIDE beyond Java.

We proceed in two steps. First, we analyze the underlying principles behind the rules and exceptions we found for Java to derive a general model. Second, we describe our approach to automate the process of extending CIDE for new languages with minimal human effort.

4.1 Generalizing Safe Decomposition Rules

To decompose a software artifact with CIDE, we need an underlying structure, so that transformations to remove colored fragments are unable to produce syntax errors. 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. We develop a *generalized model for CIDE (gCIDE model)* that represents the necessary structure. The full gCIDE model is depicted in Figure 5.

Basics. The Subtree Rule can easily be applied to arbitrary tree structures: whenever a tree node is colored, its children are colored as well. In the gCIDE model, this tree structure is represented by structural elements that can have other structural elements as children and have exactly one parent each (except for the root that represents the whole artifact). Because CIDE should later map a user selection to this underlying structure it is also necessary to store the location of the structural element inside the artifact (e.g., by offset

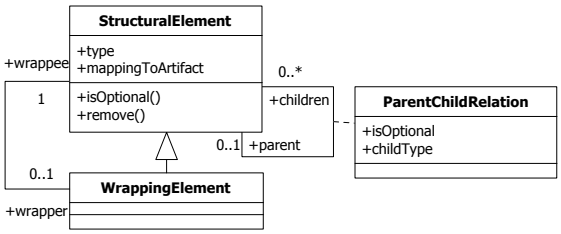

Figure 5. gCIDE Model

and length, 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 is the Java Language Specification [16], in XML the allowed structure is specified by a W3C recommendation [6], for other artifact structures such specification either exists or must be formulated to determine which (removal) transformations on the structure are safe. In the gCIDE model, we can determine for every element whether it is optional, by the *isOptional* attribute of the relationship to its parent.

Wrappers and Types. For the wrapper exception to the Subtree Rule, i.e., that we could remove a *try-catch* statement without removing the statements it surrounds, we introduce the notion of a wrapping element in our model. A wrapping element, when removed, is replaced by a specific child element it wraps around. In the gCIDE model, a wrapping element is a special case of a structural element and specifies exactly one child element it wraps around (implications of allowing to wrap around multiple child elements are discussed below).

However, wrapping elements cannot be placed at arbitrary places or wrap around arbitrary elements. For example, in Java a class cannot wrap around a method so that the class is replaced by this method if removed, because 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. Each structural element belongs to a type, and there is a subtype relation on those types. For example, structural elements representing *try-catch* or method invocations share the same supertype *Statement*. This means that in all places in which a structural element of the type *Statement* is allowed (e.g., inside a block), all structural elements with subtypes of statements can be used. In the gCIDE model, types are introduced for structural elements, and each parent-child relationship specifies the accepted type.

With types for structural elements, we can easily define when wrappers are allowed. A structural element can wrap around another element, if the wrapped element is compatible with the type expected for the wrapper. Specifically, the wrapper itself is a subtype of some expected *childType* from its parent element (e.g., an *try-catch-statement* in a block is a subtype from the expected statement), and the wrapped element must be a subtype of *childType* as well (e.g., it must also be a statement, not an *expression* or *catch-block*).

Note, it is conceptually also possible to model structural elements that wrap around multiple elements. Handling such wrappers is possible under some conditions, but requires 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 the wrapper with multiple elements. Alternatively, more complex custom transformations could be specified. To keep the model simple, we allow wrappers to wrap a single element only.

4.2 Automating Language Plug-in Creation

In the previous sections, we generalized the lessons learned from Java to an abstract model, in which software artifacts are represented in a tree structure. With the gCIDE model we can now generalize CIDE for different artifact types. For that, 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, i.e., create the structural elements for the target language and fill 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. Figure 3, Step 2 and 4) from the implementation of specific target languages as plug-ins. The new architecture of CIDE with a general framework and several language extensions is depicted in Figure 6.

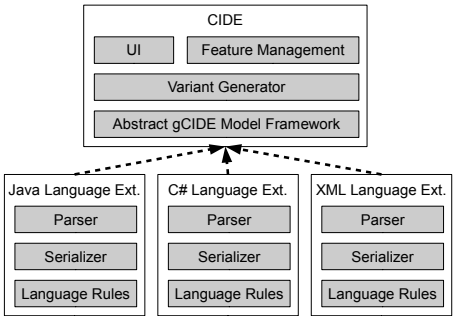

Figure 6. Language Plug-ins in CIDE – Architecture

However, the effort to create language plug-ins for CIDE is high. First, we need a parser for each language that transforms the artifact into the tree structure (cf. Figure 3, Step 1). Second, we need to implement serialization for each lan-

guage, which writes the transformed structure tree back to an artifact (cf. Figure 3, Step 5). Finally, and most importantly, we need to define the rules for each specific language, e.g., we need to define which structural elements are optional or wrappers, so that transformations always transform valid trees into new valid trees.

A straightforward way to implement language plug-ins is to bridge the internal structures of an existing open compiler for that language to the gCIDE model. For example, we could bridge the AST from the Eclipse Java framework and thus reuse Eclipse's parser. However, open industrial-strength compilers that can be reused are not available for a variety of languages. Furthermore, implementing the bridge might require considerable effort. Instead, we pursue an approach, in which we can uniformly *generate* language plug-ins for arbitrary languages.

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. Existing parser generators can generate a parser from the language's grammar. Some parser generators can also create a pretty printer that can be used for serialization. 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 and (2) which elements are optional. That is, we can use the grammar of a target language to generate a language plug-in, instead of implementing it from scratch.

To generate language plug-ins for CIDE from grammar specifications, we built our own tools, because common parser generators (e.g., JavaCC or ANTLR) do not propagate sufficient information from the grammar to the created tree structure. For example, from the parse tree that JavaCC or ANTLR generate, we can not determine which elements are optional. Therefore, we decided to define our own grammars specification language called *gCIDE grammar* and to build an own tool called *astgen* to generate *LL(k)* parsers,[2] serializers, and trees with all information required by the gCIDE model. (Technically, the gCIDE 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 gCIDE format. From this Java grammar, *astgen* generates all required parts for a Java language plug-in.)

gCIDE Grammar Basics. The gCIDE grammar specification language is close to Backus-Naur form. It consists of production rules with one or more choices each, which again contain a sequence of elements each. Elements can either refer to other productions, be tokens or some special annotations for *astgen*. From a given grammar, elements are recognized as optional, when written square brackets, 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>)* ;
```

Figure 7. gCIDE grammar example

From a given grammar for a target language, *astgen* generates the language plug-in, i.e., a parser that builds a tree structure for a given artifact in that language, and a 'pretty printer' that serializes 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, i.e., 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 and can be colored (there can be any number of type declarations in a compilation unit, Line 1). Inside the type declaration the first identifier and the class body are mandatory and can not be colored, but the second identifier and the 'implements' list are optional (Line 2). Also members are optional and can be colored (Line 3).

Concrete Syntax vs. Abstract Syntax. Generating a tree structure from a grammar for a target language results in a *Concrete Syntax Tree (CST)*. The tree contains those elements that are required for parsing. However, a CST does not necessarily reflect the abstract syntax of the language, and mapping the CST to the gCIDE model (instead of the 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 type declaration, but inside the list the first entry is mandatory, due to special parsing requirements for the separating commas. Using the CST, the first entry cannot be colored individually, although all entries are optional elements of a list in the abstract syntax.

To ensure the full flexibility of the abstract syntax, it is necessary to transform the CST generated by a standard parser into an AST. For serialization, the same transforma-

[2] For parser generation, we internally reuse *JavaCC*.

tion must be performed backward 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 concept of Wile who used an extended grammar specification language to derive the abstract syntax from an annotated grammar file [46]. Wile proposes 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 the generated parser can directly build an AST. Wile further proposes 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: IDENTIFIER ^ ",";*', 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 list and build the AST accordingly. We adopted this concept and implemented those extensions Wile proposed in our gCIDE grammar specification language that are relevant for our case studies. This way, we can generate a parser that creates structural elements based on the target language's abstract syntax from a grammar file. No manual mapping between CST and AST is required, all further transformations to remove colored fragments can be directly performed on the AST of the artifact.

Wrappers and Other Exceptions. Wrappers are not automatically recognized from the grammar for a target language, but a language expert has to decide where wrappers make sense. Though, our tools could analyze where wrappers are possible according to the typing rules sketched in Section 4.1, wrappers should only be used where the additional flexibility is needed. Otherwise, it would still enforce syntactic correctness, but be harder to use.[3]

In the gCIDE grammar specification language, we employ additional annotations 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 be defined with further annotations. For the concrete syntax, we defer the interested reader to the language description at the CIDE web site.

[3]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, coloring a whole class except an inner class, usually does not make any sense. Offering such flexibility is only confusing to the developer; workarounds as preliminary refactorings are much easier to use.

5 Supported Languages

Using our approach, we generated language plug-ins for a variety of languages, which all share CIDE's guarantee of syntactic correctness. Creating gCIDE grammars was mostly simple and typically required only few hours each (including tests), because we could often reuse or adapt existing grammar specifications. In the following, we list the languages that are currently supported and briefly discuss our experiences. We selected these languages for different reasons, some because they were required for a industrial project or case study, some to compare languages from different paradigms. All listed languages were generated from a gCIDE grammar and can be downloaded from the CIDE web site.

Featherweight Java. A first, very simple language is Featherweight Java, a core functional subset of the Java language that mimics the Java module system. The handwritten grammar file consists only of few production rules, yet it is sufficient to allow users to color individual methods, fields or parameters. We found no need for annotations like wrappers, because of the language's simplicity.

Java. Next, we reimplemented a language plug-in for Java 1.5, this time not using Eclipse's Java framework, but generating parser, serializer, and AST using gCIDE. We used a grammar from the JavaCC repository as starting point, which made the creation of an initial version very simple. Furthermore, using gCIDE annotations, we added seven wrappers that were manually implemented in the original implementation. Regarding syntactic correctness and flexibility, the generated version is equivalent to the original CIDE implementation based on Eclipse's Java framework.

C. Although parsing C code is a very complex task – for example the parser often depends on types recognized earlier during parsing to determine how to parse a code fragment – creating a gCIDE grammar for C based on an existing grammar in the JavaCC format was straightforward. With this grammar, we could generate a CIDE language plug-in for C, in which users can color individual functions, declarations, statements, or parameters.

Unfortunately, the C parser works only on C code that has already been preprocessed, i.e., 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 [37] or refactoring tools [15] 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 following semicolon, functions by the typical pattern for return type and parameter declarations. Preprocessor directives are recognized as part of the language, as long as they are used within certain limitations (e.g. *'#ifdef'* may only encapsulate whole statements or functions). With this pseudo-parse we are able to use CIDE on C projects.

C#. In contrast to C, C# does not use a preprocessor and can be parsed completely. We converted an existing ANTLR grammar for C# into the gCIDE format and added some annotations for wrappers. In CIDE, C# can be used very similar to Java, i.e., users can color classes, methods, statements, parameters, and so on, safely based on the underlying AST.

Haskell. Next, we created a language plug-in for Haskell, a functional language that has a very different syntax compared to Java. The language is very complex and fully parsing all expressions and patterns requires a high effort. Further, writing a full parser would have required an extensive transformation from an existing *LALR* grammar [25] to our required *LL(K)* format. Therefore, we again used a pseudo-parser approach (reusing a pseudo-grammar from a Haskell plug-in for Eclipse[4]) that skips over certain complex productions of the source code like expressions and pattern and only parses the main structure.

ECMAScript. Next, we generated a language plug-in for ECMAScript (better known as JavaScript or JScript), a dynamically typed, prototype based script language. The gCIDE grammar was derived from an existing JavaCC grammar. Users can safely color functions, statements, parameters, etc. as described by the language's grammar.

JavaCC, ANTLR, Bali. Yet, another different group of artifacts are grammar files in diverse formats. For example, in an SPL of Java compilers a feature might extend the grammar of Java 1.4 with generics. We generated language plug-ins for JavaCC, ANTLR, and Bali artifacts, in which users can color productions or optional parts therein. Wrappers are used for modifiers like multipliers or square brackets (i.e., it is possible to color only the square brackets in the production *'A: B [C]'* making C mandatory in variants in which the feature is not included). This allows us to safely decompose language grammars, which can be part of a product line.

[4] `http://eclipsefp.sourceforge.net/`

XML, HTML, XHTML. Finally, we created language plug-ins for XML and HTML files. We created a gCIDE grammar for most parts of XML based on the language specification published by the W3C [6]; for HTML we used one of many existing JavaCC grammars. In HTML, this allows us to color structural elements like headings, paragraphs, lists and so on safely. In XML, nodes and attributes can be colored in a way that guarantees that every variant is syntactically correct, i.e., *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 may require additional information. Note, DTD is a meta-grammar itself that can be used for XML documents instead of gCIDE. We implemented a prototype transformation tool *dtdgen*, that converts a DTD into a gCIDE grammar, as a proof of concept. This way, we generated a parser specifically for XHTML (version '1.0 strict') and some other specific XML-based languages. However, in the long-run, we aim at a direct transformation from a DTD or XML Schema to the gCIDE model using an off-the-shelf XML parser.

6 Flexibility vs. Safety

During development and testing, we found that two properties are in conflict: flexibility and safety. By imposing a structure on a source code artifact, we can guarantee syntactic correctness, but at the same time, we reduce flexibility of decomposition. For example, compared to a decomposition using the C/C++ preprocessor that works on token level[5], CIDE allows only to color optional elements. Therefore, using a preprocessor, it is possible for a method to have two alternative return types, but in CIDE a return type cannot be colored independently if it is mandatory by the language 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 color 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, ECMAScript artifacts only contain a list of statements or function declarations, grammar artifacts only contain a list of productions 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

[5] To be precise the C/C++ preprocessor works on lines of source code. However, due to the code layout flexibility in most languages, it can usually be used as token-based decomposition.

the future?

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 such grammar, every character is optional with respect to the document, i.e., every single character in this document can be colored independently like with preprocessors. This shows that a required structure is not a limitation of our approach, in contrast to other (especially compositional) approaches that assume a tree structure with unordered named elements [5, 4, 1, 3]. Even if no structure is available, as in the 'README.txt' file, we can still use the same tool uniformly to color this file in line with 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 the syntax checks. It restricts the possible parts of the artifact that can be colored and enforces a 'reasonable' decomposition.

In this context, the pseudo-parsing approach presented for C and Haskell artifacts in Section 5 is an interesting case. This approach 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 and because of the complexity of Haskell). 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 inner fragments that are ignored by the parser. First, we can regard these fragments as a single mandatory node each, i.e., there is no substructure inside a statement (alternative A, used in Haskell). Alternatively, we can parse these fragments with a dummy grammar as a list of optional tokens or characters (alternative B, used in C). 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 with more flexibility guarantees syntactic correctness only for the recognized parts but not for their inner structure.

In Figure 8, we visualize the relative differences in safety and flexibility of all approaches. For decomposition based on a concrete syntax tree or an abstract syntax tree, we can guarantee syntactic correctness, while the AST provides more flexibility. 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

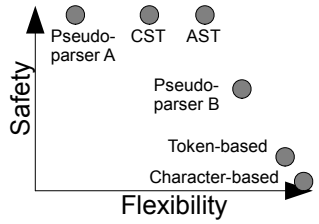

Figure 8. Safety vs. Flexibility

decomposition – as with the dummy grammar or *'#ifdef'* preprocessors – provides the most flexibility (every single character or token can be colored) 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.

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

7 Toward Checking for Language Semantics

In the previous section, we have shown how CIDE guarantees that every variant is syntactically correct, independent of the language. This is helpful to prevent errors when decomposing legacy applications. Furthermore, this provides a structural approach to decomposition that does not directly use the source code representation but an abstraction thereof. Still, many other kinds of errors are possible. In many languages syntactic correctness is not sufficient, but artifacts have additionally to adhere to *language semantics* (not be be confused with program semantics) that are not covered by the syntax and thus cannot be generated from a grammar.

A straightforward example comes from programming languages like Java, that must not only be parseable, but also well-typed. While it is syntactically correct to remove a method from a class even though it is still called from another method, the generated variant would cause a compile-time error because Java's type checks fail. Another example comes from UML models, in which the syntax guarantees that every class element has a name and that interfaces do not have attributes, but errors like dangling associations cannot be recognized from the grammar.

Language specifications as the Java Language Specification [16] or the Haskell 98 Language Report [25] describe both, the language syntax and the language semantics. Everything that can be matched by a grammar (in our case an *LL(k)* grammar) belongs to the language syntax, all addi-

tional constraints like type conditions or other integrity rules are specified as language semantics. For example, the Java Language Specification defines *'It is a compile-time error for the body of a class to declare as members two methods with override-equivalent signatures.'* [16, Sec. 8.4], a condition that cannot be covered by Java's context-free grammar.[6]

CIDE, as described so far covers only the language syntax, not language semantics. Ideally, an SPL tool should be able to verify both in order to avoid errors without compiling all variants. To make matters worse, as SPLs are typically implemented with artifacts in multiple languages, it is not sufficient to check artifacts from a single language, but artifacts from different languages may reference each other and must be consistent. This raises the question of *inter-language semantics* and whether it can be checked automatically for all variants of an SPL [17]. A typical example is a Web service whose interface is described with an WSDL[7] file in XML format and implemented with a programming language like Java. In all variants, the Web service implementation should be consistent with its description.

Language semantics are far more difficult to check than language syntax. They are often only specified informally – as in the Java Language Specification – and are typically implemented in the front-end of a reference compiler. This makes it very difficult to analyze language semantics formally and *guarantee* safety for all variants. Especially inter-language semantics are rarely formalized. Only for some languages, language semantics are specified with formal mechanisms that allow to formally proof safety, e.g., language semantics in UML models are specified by OCL constraints [12] and language semantics for some languages like Featherweight Java are described with a calculus [20]. However, for the majority of commonly used artifacts no such formalization exists. Previous approaches that ensure safety for all variants therefore used approximations and modeled only the most important conditions [19, 42] and so did we in the original Java version of CIDE.

In the following, we discuss what is required to check language semantics for all variants in CIDE. Again, we start with our existing approach to check language semantics of Java, then we generalize it to language semantics of other artifact languages and to intra-language semantics.

[6]The difference between language syntax and language semantics can be blurred for some languages and depends on the implementation. For example to parse XHTML, we can create a parser that parses only valid XHTML artifacts (as we did in Section 5). This way, XHTML can be specified by syntax without additional language semantics. Alternatively, we can parse XHTML as XML (based on the XML grammar) and check all structural conditions (e.g., a document must consist of header and body) as language semantics. Taking this to extremes, it would also be possible to parse Java using the dummy grammar and model the syntax as language semantics.

[7]Web Services Description Language, `http://www.w3.org/TR/wsdl`

7.1 Checking Language Semantics for Java

Already early on, we found it very useful to check several language semantics in Java to ensure a reasonable coloring and safety for all variants. Actually, these checks were very useful for code exploration as well, because they alerted us of incomplete colorings of features. The effort of implementing checks for all possible compiler errors from the Java Language Specification [16] was too high, therefore we focused on the – in our experience – most frequent causes of compiler errors in variants: the inability to resolve methods, fields, or types and mismatching method signatures caused by coloring parameters. If we implemented all checks, we could guarantee that all variants can be compiled without actually generating them. With the currently implemented checks we can still detect the most common problems.

We exemplify the mechanics of such checks with the condition in Java that method invocations must be statically resolvable [16, Sec. 15.12]. Our current solution (others might be possible) that guarantees that all variants meet this condition is straightforward. CIDE resolves all invocations of methods and determines the according method declarations. For each pair of invocation and declaration, CIDE checks that the method invocation is colored with the same colors as the method declaration (or with a superset). For example, consider that the method declaration is colored with feature *F*. If the invocation of this method is not colored, then all variants without feature *F* result in dangling method references and, thus, compiler errors. However, if the method invocation is also colored with feature *F* (and possibly additional features) then the invocation only occurs in variants where the declaration exists as well. If this simple check succeeds for all pairs of method invocation and declaration, then all variants are free of compilation errors caused by unresolvable method invocations.

This approach breaks down an apparently complex check – ensure that all invocations in all variants can be resolved – to a simple set comparison of colors on two Java elements. We used the same mechanics for a number of conditions we implemented in CIDE from the Java Language Specification: (1) Every field access must be colored with at least the same colors as the referenced field declaration. (2) Every access to a local variable must be colored with at least the same colors as the referenced local variable declaration. (3) Every reference to a type (e.g. in a variable declaration or import declaration) must be colored with at least the same colors as the type declaration. (4) Every parameter in a method call must have exactly the same colors as the parameter in the method declaration.[8]

[8]Note, this rule is special, because parameters that are declared but not used in the call result in an error, in contrast to types or methods that are declared but not used. Also, only the directly declared colors have to be equal, colors inherited via the *Subtree Rule* follow the other conditions above.

The implemented conditions are not complete, but cover the most common violation of Java's language semantics. To fully ensure language semantics in all variants, it is necessary implement all conditions for Java's language semantics, including more complex ones like *'a concrete classes must implement or inherit all methods from superinterfaces'* [16, Sec. 8.1.1.1] that have to take into account combinations of colors for methods, interface declarations, and parameters. Nevertheless, it is possible to balance flexibility and effort, e.g., by restricting the flexibility of the language's gCIDE model (e.g. not allow to color modifiers or superinterfaces from the implements clause) so that less checks have to be implemented.

7.2 Generalizing Language Semantic Checks

After generalizing the tool so that we can language-independently guarantee syntactic correctness for all variants, it is natural to ask for language-independent checks for language semantics. It seems that we cannot offer intra-language checks for every language. For example, for dynamically typed languages like ECMAScript (JavaScript), method invocations cannot be resolved by static code analysis, thus language semantics cannot be formalized and at most be tested based on heuristics. On the other hand, for grammar specification languages like JavaCC or Bali, it is surprisingly simple to validate colors: non-terminals reference other productions and semantics can be checked with a single simple rule: *non-terminals must be colored with at least the same colors as the production they reference*.

We observed in Java and other languages that most conditions are checked by comparing references between two elements. Thus, we need two components, first a *Resolver* that finds references between elements (usually with the help of a static type system) and second a *Color Checker* that defines and checks conditions for color on those references. Many conditions can be defined by simple set operations on colors taken from a library of common checks. However, more complex conditions can be modeled using a more complex logic, rule, or specification languages like propositional logics [42], OCL [45] or Alloy [21], if necessary.

For future work, we envision to model language semantics in a formal language (similar to language semantics in models that can be specified formally with OCL) that can be automatically translated into checks on colors. We will start with Featherweight Java, for which a formalization in form of a calculus already exists, and languages like grammars for which semantics are obvious.

In SPLs that consist of artifacts in different languages, the challenge arises that references between those languages should be checked as well. Given a formal model for intra-language semantics, it will also be possible to define inter-language semantics on top with the same mechanics. The only difference is that the Resolver needs to detect references between language elements from different artifact types (e.g., the reference from a WSDL interface specification to a C# method declaration). If a polylingual type system does not exist (what is typically the case), we need to implement the resolver for specific combinations of artifact types individually. Once references are resolved, checks on colors can be performed with the same mechanisms as for intra-language semantics.

8 Case Studies

In the previous sections, we described how we created a grammar specification language and a tool suite for generating language plug-ins. All language plug-ins described in Section 5 are fully implemented and were tested with many artifacts. To show that CIDE is actually useful for developing SPLs and to show how important handling multiple artifact languages is for SPL development, we additionally conducted a series of case studies. For that purpose, we revisited some SPLs developed with other product line technologies earlier and developed some new projects. All case studies are available on the CIDE web site, except for the last one that we have to disclose to protect intellectual properties of our partners.

Berkeley DB and GPL. First, we review two Java case studies we published in prior work [27]: the 84 000 LOC embedded database engine *Berkeley DB* decomposed into 38 features and a small SPL of graph algorithms (Graph Product Line, short *GPL*) with 14 features proposed as benchmark for SPL technologies. In both case studies, the Java version of CIDE was applied to color features and has shown benefits in granularity what made the decomposition simpler and faster. The generalization of CIDE now offers the opportunity not only to decompose the Java code, but also the accompanying documentation. Consequentially, in GPL, we additionally decomposed the documentation in form of a single XHTML file, in that its basic mechanisms and all algorithms were described. Overall, we colored 145 of 218 lines of XHTML code. The colored fragments are typically headings, paragraphs, or rows from a table. When a developer now creates a variant of GPL, CIDE creates not only the according Java code but also a documentation that exactly describes those algorithms and properties that are applicable in that specific variant. Similarly, we also decomposed the HTML documentation of Berkeley DB coloring fragments from 14 of the 56 HTML files of the *'Getting Started Guide'*, e.g., removing the description of the transaction subsystem or several background threads for variants in which they are not included. In Berkeley DB, we also found that checks for (informal) HTML language semantics would be useful, in that hyperlinks are checked for dead links in all variants, as

we had to remove several links from the table of contents and other parts of the document and it is easy to miss one.

Large Scale Multi-Representation Decomposition. Second, we reimplemented the decomposition of the AHEAD tool suite with CIDE, which two of the authors did manually in prior work with the compositional approach [43]. The AHEAD tool suite is a collection of tools for feature-base program synthesis which over time has grown to 24 different tools with over 200 000 lines of Java code. In addition to code, there are makefiles, regression tests, documentation, and program specification, "all of which were intimately intertwined into an integrated whole" [43]. To better manage the tool suite and to be able to remove or replace tools a feature-oriented decomposition into 13 features for different tools was performed. Especially build files and documentation were decomposed with special XML tools. With CIDE, we redid the whole refactoring within less than 2 hours[9], by just coloring fragments inside different artifacts.

Customizable SQL Parser. Third, we applied the generalized version of CIDE to our ongoing work on highly customizable SQL parsers as part of a project for tailor-made data management in embedded systems [40, 36]. In this project, we decomposed the grammar of SQL:2003 to include only those operators that are needed for a specific application. For example, if grouping data is not required, it can be removed from the grammar, the resulting parser, the query optimizer, and from the list of operations in the database engine. This is a typical polylingual problem, because the grammar is written with the ANTLR grammar specification language but the optimizer and database operations are written in C or Java. A feature like *Grouping* crosscuts different artifacts and parts of the database system. In the context of this project, we prototypically applied CIDE to decompose an SQL grammar subset and intend to also decompose the optimizer and operations that are still to be developed.

Arithmetic SPL in Haskell. Fourth, we created an SPL of Haskell programs for arithmetic expression evaluation. Currently, the SPL contains 4 features: *base*, *variables*, *no-variables*, and *if-then-else*. With CIDE, we colored several declarations that are safely removed for variants. We could successfully generate and run all of the four variants of the program. Though this SPL is still small, CIDE's ability to adapt to new languages by only specifying a language grammar opened the possibility to confer the SPL concept from Java or C that are typically used in SPL research to rather uncommon languages. CIDE with its Haskell language plug-in will be at the heart of our future research on functional software product lines.

Industrial Project in C. Finally, CIDE was used on an industrial project written in C to decompose the embedded software of a gas boiler application into features.[10] The software for this embedded system existed in many variants for different hardware configurations (analog, digital, atmospheric, or bitermic boiler types; 24 KW or 30 KW power capacity; propane, butane, or solar energy; different interfaces). Decomposing the software into an SPL was a natural endeavor to ease maintainability and manage the increasing amount of functionality. The general results of decomposing the software into features will be published in a separate paper, here we focus only on CIDE's impact. This project and the experience with manual decomposition was one of the key motivations to generalize CIDE for programming languages beyond Java. After only few minor source code changes toward a more disciplined usage of preprocessor directives, we were able to parse the source code with our pseudo parser and color code fragments reflecting the different hardware configurations. Using CIDE, it is now possible to safely generate different variants of the water boiler system, that can be compiled and used.

9 Related Work

Language-independent SPL Tools. The principle of uniformity that claims that all artifacts during SPL development should be handled uniformly by the SPL technology was first introduced by Batory at al. [5]. Traditional SPL technologies like preprocessors as in C/C++, Frames/XVCL [23], or tools like *Gears* [30] provide this uniformity as they annotate code at the level of text files. They do not require any additional structure and thus can be used on arbitrary artifacts. At the same time, correctness must be tested by generating and compiling individual variants.

The AHEAD tool suite implements the principle of uniformity by providing tools for several artifact languages, e.g., Java, XML, and grammars. Although, the tool suite provides an extensible infrastructure to create tools for new languages, extensions were usually ad-hoc and implemented additional tools from scratch. Recent research by Apel at al. focused on this problem by analyzing the underlying principles of feature composition and generalizing it in a formal framework [3, 4]. They build a tool called *FSTComposer* that can handle several artifacts uniformly, for which only the parser has to be written for each language while the composition is done by a general framework (we are currently synchronizing work to reuse gCIDE grammars also for FSTComposer).

[9] Admittedly, we benefited from the fact that no code exploration was necessary; feature code was apparent from the first decomposition.

[10] The interested reader may contact the third author Salvador Trujillo for details about this case study.

Nevertheless, the AHEAD tool suite and the FSTComposer are both tools that use the compositional approach and conceptually can not handle the fine granularity often needed when decomposing legacy applications [27].

Related Tools. The concept of using the representation layer to show additional information without obfuscating the source code as in CIDE was already used by several development environments. Recent examples are *presentation extension* [13], *AspectBrowser* [18] and *Spotlight* [10].

IDEs for visual programming and intentional programming abstract from traditional code representations and store code in internal tree formats close to structural elements in the gCIDE model. The idea of storing program code in databases to allow flexible queries to create different views on the code goes back to Linton [31]. Modern examples are *effective views* [22] and the *Domain Workbench* [37, 38], that store code in internal tree structures.

Further, there are several tools to annotate source code with features. However, such work usually does not aim at SPLs or code transformation but at code exploration or at making concerns or features explicit. For example, Robillard and Murphy suggested *concern graphs* where developers can collect methods belonging to a feature in an external window [35]. Work on *visual separation of concerns* extends this and provides aggregated views on the source code by features [7], however, they do not ensure that this view is consistent or syntactically correct. Furthermore, the *AspectBrowser* [18] uses pattern expressions or queries to find code fragments belonging to a certain feature. It works on character level and supports arbitrary artifacts, but does not ensure correctness or consistency.

Closest to our work is *fmp2rsm*, a tool for annotating and generating variants of UML models by Czarnecki and Antkiewicz [11]. In this tool, UML elements of a superimposed model are annotated with 'presence conditions', similar to colors in CIDE. Variants are created by removing elements whose presence condition does not evaluate to true for a given feature selection. An interesting concept which we might also consider for future versions of CIDE are meta-expressions that allow alternative values for a model attribute. An important distinction between *fmp2rsm* and CIDE is, that in *fmp2rsm* arbitrary propositional formulas are attached as presence conditions, whereas CIDE only allows conjunctions of features, i.e., a set of colors. This restriction allows a much simpler user interface in that colors are simply blended (expressions like 'feature A or feature B and not feature C' cannot be displayed with the color metaphor). Whether this simplification significantly restricts expressiveness is an open research question.

Checking Language Semantics. An influential approach proposed by Huang at al. ensures correct language semantics for Java code generated by their tool SafeGen [19]. Their tool is used for meta-programming in general, not as SPL technology. Using first-order logics and theorem provers, they check whether generators written in their confined meta-language (with selection and iteration operators) produce syntactically correct and well-typed output for arbitrary Java input. A selection of language semantics are formulated using first-order logics and checked for a given generator, but checks do not cover all language semantics and are only available for Java. This approach was starting point for checking language semantics in the following two SPL tools.

Based on their SPL tool *fmp2rsm*, Czarnecki and Pietroszek proposed an approach to check language semantics of their models against OCL constraints for all variants [12]. Language semantics for models can be specified formally using OCL constraints. Their tool now transforms the presence conditions and OCL constraints into a propositional formula that can be solved by an off-the-shelf SAT solver. By restricting their approach to models specified in MOF, they can ensure that all variants adhere to the model syntax and the *complete* model semantics (all variants are *well-formed* in the MOF terminology). Again the decision to use arbitrary propositional formulas increases flexibility but also complexity for evaluation of their approach, compared to simple set operations in CIDE.

A different approach to ensure language semantics in all variants of an SPL called *safe composition* was proposed by Thaker at al. [42]. They analyze language semantics of Jak, a Java dialect for feature-oriented programming. In Jak, features are implemented in distinct feature modules and composed to generate a variant. Syntactic correctness of all variants can be ensured by the composition mechanism, i.e., when all feature modules are syntactically correct the composition tool can only create variants with correct syntax. To check language semantics, they identify six constraints that need to be satisfied, including method invocations as discussed above. They analyze these constraints for all features individually, and reason about the composition of all variants using propositional formulas and a SAT solver. This approach is used only for Jak and XML and there is no proof of completeness that all language semantics are covered.

Both SPL tools check correctness against a feature model that may restrict certain variants with domain constraints. In CIDE, we take a more general approach, in that all variants are checked, even those that cannot be generated because of domain constraints. Instead, domain constraints must be reflected by colors in the code. While this does not add any security or expressiveness, it simplifies the reasoning and reduces most semantic checks to set comparisons. Further, we claim that colored programs are simpler to understand, as the colors already reflect domain constraints and no external model is needed for verification. A detailed comparison of both mechanisms is outside the scope of this paper, but will

be part of our formal semantic checks in future work.

Software Merging. A similar problem as in CIDE for safely removing code fragments for variants occurs when *merging* artifacts, e.g., in version control systems. While simple tools like *diff* operate language-independently on simple text files, more advanced tools include language syntax or even language semantics from selected languages to improve accuracy, i.e., to detect more faults before merging two artifacts. Some tools even balance between safety and performance using a pseudo-parsing approach. In [33], Mens gives a comprehensive overview on software merging and the concepts used.

10 Conclusion

Software product lines (SPL) usually consist of artifacts written in different languages. To handle different artifacts uniformly ('principle of uniformity'), current SPL technologies either use an approach that is so general that it works for arbitrary artifacts, but can easily introduce subtle errors for some variants; or they provide specialized tools for a low number of languages. Syntactic errors that only occur in certain variants of the SPL are a serious problem, as the exploding number of variants makes a manual testing by generating and compiling each variant unfeasible.

We have shown how we can extend CIDE, a tool for decomposing systems into features, to arbitrary languages by abstracting from the concrete textual representation in a file. CIDE uses the abstract structure of the language and can therefore ensure that all created variants are syntactically correct. We have shown, how we can generalize the concepts from Java to a general model for arbitrary languages, so that CIDE can be used uniformly for SPL development. In a further step, we even automated the process of creating language plug-ins from annotated grammar files, so that extending CIDE (including its guarantee for syntacical correctness) for new languages requires minimal human effort.

While CIDE only checks for syntactic correctness, further errors may arise in some variants due to violations to the language semantics. Moreover, even inconsistency between artifacts written in different languages (intra-language semantics) can occur. While due to missing formalization, checking language semantics is more difficult, we have presented first results and a perspective for further work on an automated language-independent check of (inter-)language semantics in CIDE.

We have shown CIDE's applicability by generating plug-ins for a series of code and non-code languages including Java, C, C#, Haskell, ECMAScript, JavaCC, and XML. We have further shown CIDE's usability for concrete problems in several different case studies that consist of artifacts written in many different languages.

Acknowledgments. We thank Peter Kim for fruitful comments and discussions on earlier drafts of this paper. We further thank Marko Rosenmüller and Norbert Siegmund for their help and patience when developing and testing the C grammar, Armin Größlinger for providing programs for the Haskell case study, and Sagar Sunkle for releasing the decomposed SQL grammars.

References

[1] F. I. Anfurrutia, O. Diaz, and S. Trujillo. On the Refinement of XML. In *Proc. International Conference Web Engineering (ICWE)*, Como, Italy, July 2007. Springer.

[2] Sven Apel, Thomas Leich, and Gunter Saake. Aspectual Feature Modules. In *IEEE Transactions on Software Engineering*, 2008. Online first.

[3] Sven Apel and Christian Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *ETAPS International Symposium on Software Composition*, 2008.

[4] Sven Apel, Christian Lengauer, Don Batory, Bernhard Möller, and Christian Kästner. An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau, 2007.

[5] Don Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering*, 30(6):355–371, 2004.

[6] Tim Bray et al. Extensible Markup Language (XML) 1.1 (Second Edition). W3C Recommendation, W3C, August 2006.

[7] Mark Chu-Carroll, James Wright, and Annie Ying. Visual Separation of Concerns through Multidimensional Program Storage. In *Proceedings of the International Conference Aspect-Oriented Software Development (AOSD)*, pages 188–197, New York, NY, USA, 2003. ACM Press.

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

[9] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[10] David Coppit, Robert Painter, and Meghan Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proceedings of the International Conference on Software*

Engineering (ICSE), pages 754–757, Washington, DC, USA, 2007. IEEE Computer Society.

[11] Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, pages 422–437, 2005.

[12] Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-based Model Templates against well-formedness OCL Constraints. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, pages 211–220, New York, NY, USA, 2006. ACM Press.

[13] Andrew Eisenberg and Gregor Kiczales. Expressive Programs through Presentation Extension. In *Proceedings of the International Conference Aspect-Oriented Software Development (AOSD)*, pages 73–84, New York, NY, USA, 2007. ACM Press.

[14] Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java™ Language Specification*. The Java™ Series. Addison-Wesley Professional, 3 edition, 2005.

[17] Mark Grechanik, Don Batory, and Dewayne Perry. Design of Large-Scale Polylingual Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 357–366, Washington, DC, USA, 2004. IEEE Computer Society.

[18] William Griswold, Jimmy Yuan, and Yoshikiyo Kato. Exploiting the Map Metaphor in a Tool for Software Evolution. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 265–274. IEEE Computer Society, 2001.

[19] Shan Huang, David Zook, and Yannis Smaragdakis. Statically Safe Program Generation with SafeGen. In Robert Glück and Michael R. Lowry, editors, *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 309–326. Springer, 2005.

[20] Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[21] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, Cambridge, MA, USA, 2006.

[22] Doug Janzen and Kris De Volder. Programming with Crosscutting Effective Views. In *Proceedings of the European Conference Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 195–218. Springer, 2004.

[23] Stan Jarzabek, Paul Bassett, Hongyu Zhang, and Weishan Zhang. XVCL: XML-based Variant Configuration Language. In *Proceedings of the International Conference on Software Engineering (ICSE)*, page 810, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[24] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[25] Simon P. Jones, editor. *Haskell 98 Language and Libraries – The Revised Report*. Cambridge University Press, 2003.

[26] Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, 2007.

[27] Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2008. to appear.

[28] Christian Kästner, Martin Kuhlemann, and Don Batory. Automating Feature-Oriented Refactoring of Legacy Applications. In *Poster presented at Europ. Conference Object-Oriented Programming*, July 2007.

[29] Gregor Kiczales et al. Aspect-Oriented Programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the European Conference Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, July 1997.

[30] Charles Krueger. Easing the Transition to Software Mass Customization. In *Proceedings of the International Workshop on Software Product-Family Eng.*, pages 282–293, London, UK, 2002. Springer-Verlag.

[31] Mark Linton. Implementing relational views of programs. *SIGPLAN Not.*, 19(5):132–140, 1984.

[32] Jia Liu, Don Batory, and Christian Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121, 2006.

[33] Tom Mens. A State-of-the-Art Survey on Software Merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[34] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer Verlag, Secaucus, NJ, USA, 2005.

[35] Martin Robillard and Gail Murphy. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 406–416, New York, NY, USA, 2002. ACM Press.

[36] Marko Rosenmüller, Norbert Siegmund, Horst Schirmeier, Julio Sincero, Sven Apel, Thomas Leich, Olaf Spinczyk, and Gunter Saake. FAME-DBMS: Talor-made Data Management Solutions for Embedded Systems. In *EDBT Workshop on Software Engineering for Tailor-made Data Management*, pages 1–6, 2008.

[37] Charles Simonyi. The Death of Computer Languages, the Birth of Intentional Programming. In *NATO Science Committee Conference*, 1995.

[38] Charles Simonyi, Magnus Christerson, and Shane Clifford. Intentional software. In *Proceedings of the Conference Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 451–464, New York, NY, USA, 2006. ACM Press.

[39] Henry Spencer and Geoff Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *USENIX*, pages 185–198, Summer 1992.

[40] Sagar Sunkle, Martin Kuhlemann, Norbert Siegmund, Marko Rosenmüller, and Gunter Saake. Generating Highly Customizable SQL Parsers. In *EDBT Workshop on Software Engineering for Tailor-made Data Management*, March 2008.

[41] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2002.

[42] Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, pages 95–104, New York, NY, USA, 2007. ACM.

[43] Salvador Trujillo, Don Batory, and Oscar Diaz. Feature refactoring a multi-representation program into a product line. In *Proceedings of the International Conference Generative Programming and Component Engineering (GPCE)*, pages 191–200, New York, NY, USA, 2006. ACM.

[44] Engin Uzuncaova, Daniel Garcia, Sarfraz Khurshid, and Don Batory. A Specification-Based Approach to Testing Software Product Lines. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering*, pages 525–528, New York, NY, USA, 2007. ACM.

[45] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Precise Modeling With UML*. Addison Wesley, Reading, MA, USA, October 1998.

[46] David Wile. Abstract Syntax from Concrete Syntax. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 472–480, New York, NY, USA, 1997. ACM.