Language-Independent and Automated Software Composition: The FEATUREHOUSE Experience

Sven Apel, Christian Kästner, and Christian Lengauer

Abstract—Superimposition is a composition technique that has been applied successfully in many areas of software development. Although superimposition is a general-purpose concept, it has been (re)invented and implemented individually for various kinds of software artifacts. We unify languages and tools that rely on superimposition by using the language-independent model of *feature structure trees* (FSTs). On the basis of the FST model, we propose a general approach to the composition of software artifacts written in different languages. Furthermore, we offer a supporting framework and tool chain, called FEATUREHOUSE. We use attribute grammars to automate the integration of additional languages. In particular, we have integrated Java, C#, C, Haskell, Alloy, and JavaCC. A substantial number of case studies demonstrate the practicality and scalability of our approach and reveal insights into the properties that a language must have in order to be ready for superimposition. We discuss perspectives of our approach and demonstrate how we extended FEATUREHOUSE with support for XML languages (in particular, XHTML, XMI/UML, and Ant) and alternative composition approaches (in particular, aspect weaving). Rounding off our previous work, we provide here a holistic view of the FEATUREHOUSE approach based on rich experience with numerous languages and case studies and reflections on several years of research

Index Terms—FEATUREHOUSE, feature structure trees, software composition, superimposition, language independence

1 INTRODUCTION

SOFTWARE composition is the process of constructing software systems from a set of software artifacts. An *artifact* can be any kind of information that is part of or related to software, for example, code units (packages, classes, methods, etc.) or supporting documents (models, documentation, makefiles, etc.). One popular approach to software composition is superimposition. *Superimposition* is the process of composing software artifacts by merging their corresponding substructures. For example, when composing two Java files, two constituent classes with the same name, say Foo, are merged, and the result is called again Foo. The substructures of Foo are merged in turn recursively.

Superimposition has been applied successfully to the composition of class hierarchies in multiteam software development [1], the extension of distributed programs [2], [3], the implementation of collaboration-based designs [4], feature-oriented programming [5], [6], multidimensional separation of concerns [7], aspect-oriented programming [8], [9], and software component adaptation [10]. Although diverse, all these applications pursue superimposition of hierarchically organized program constructs on the basis of their nominal and structural similarities.

- *S. Apel and C. Lengauer are with the Department of Informatics and Mathematics, University of Passau, Innstr. 33, Passau 94032, Germany.*
- *C. Kästner is with the Institute for Software Research, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213.*

Manuscript received 21 Oct. 2010; revised 26 Oct. 2011; accepted 29 Nov. 2011; published online 5 Dec. 2011.
Recommended for acceptance by P. Inverardi.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2010-10-0311.
Digital Object Identifier no. 10.1109/TSE.2011.120.

It has been noted that, when composing software, not only so code artifacts—possibly written in different programming languages—have to be considered, but also noncode artifacts, for example, models, documentation, grammar files, or makefiles [6]. Thus, as a composition technique, superimposition should be applicable to a wide range of software artifacts. While there are various tools that support superimposition of code artifacts [6], [9], [11], [12], [13], [14], [15], [16], [17], [18] and noncode artifacts [6], [19], [20], [21], [22], [23], they all appear different, they are dedicated to and embedded individually in their respective host languages, and their implementation and integration require a major effort. Usually, the developers of languages and tools did not address (or realize) the general nature of superimposition. This hinders coordinated efforts to advance composition technology.

We propose a structural approach to the composition of software artifacts written in different languages, and we offer a supporting framework and tool chain, called FEATUREHOUSE. FEATUREHOUSE follows the ideas of Batory's AHEAD program generator [6] and builds on our previous work on language-independent software representation [24] and composition [25], as we will explain.

In a nutshell, FEATUREHOUSE is a framework for software composition on the basis of superimposition into which new languages can be plugged on demand. The integration of a new language, say C# or Haskell, requires only a few hours of effort, in contrast to expensive manual implementations. Technically, FEATUREHOUSE is based on three ingredients: 1) a language-independent model of software artifacts, 2) superimposition as a language-independent composition paradigm, and 3) an artifact-language specification based on attribute grammars.

0098-5589/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

We have used FEATUREHOUSE in a number of case studies to demonstrate that our approach of software composition is indeed general. In particular, we have integrated several, different languages into FEATUREHOUSE: Java, C, C#, Haskell, Alloy, JavaCC, XHTML, XMI/UML, and Ant. That is, FEATUREHOUSE can be used to compose software artifacts written in these languages. We did not need to extend the languages themselves (e.g., by introducing new syntax and extending the compiler), as would be necessary in related approaches and tools such as AHEAD [6], CaesarJ [26], Classbox/J [14], FeatureC++ [17], or Fuji [18]. This saved us a lot of tedious and error-prone implementation work.

The integration of a new language is almost entirely based on the language's grammar, plus some attributes added as annotations and some concise composition rules (usually not more than 10 to 20 lines of code). We have applied FEATUREHOUSE in the composition of over 50 software systems of different sizes (1 to 64 thousand lines of code), written in different languages (Java, C#, C, Haskell, Alloy, and JavaCC). Our studies demonstrate the practicality and scalability of our approach and tools and provide insight into mandatory and desirable properties that a language must have in order to be ready for superimposition.

In summary, we make the following contributions:

1. We propose a general approach to software composition, based on superimposition, that is applicable to a wide variety of software languages.
2. We provide a framework and tool chain for language-independent, automated software composition and report on experiments with six languages and 50 software systems.
3. We discuss experience and insight gained in our case studies, especially with regard to composition granularity, uniqueness of identifiers, and ordering of elements.
4. We present two substantial extensions of FEATUREHOUSE (support for XML-based languages and an integration of aspect weaving as an additional composition operator) which demonstrate the generality and potential of our approach.
5. We discuss perspectives of our approach, especially with regard to type checking and formal foundations.

This paper subsumes and extends our previous work on language-independent, automated software composition [27], [28], [29], [30]. For the first time, we provide a coherent and complete overview of our experience and insight gained with FEATUREHOUSE. Compared to previous work, we report on experience with a substantial number of further case studies (42 new case studies, as compared to [27]), new kinds of languages (including functional, specification, and XML-based languages), a number of extensions of our approach (e.g., a further composition operator), and perspectives that arise from our holistic and retrospective view on the FEATUREHOUSE approach. None of these contributions could have been made in earlier work, which concentrated on individual aspects, but neglected the big picture and its implications. These contributions arise from our experience and the ability to step back and look at practical applications.

2 FEATUREHOUSE

FEATUREHOUSE is a framework for software composition supported by a corresponding tool chain. It provides facilities for software composition based on a language-independent model of software artifacts and an automatic plugin mechanism for the integration of new artifact languages. FEATUREHOUSE generalizes and subsumes a previous software composition tool, called FSTCOMPOSER [25], and exceeds prior work on AHEAD in that it implements software composition language-independently.[1] The code of FEATUREHOUSE, as well as examples and all case studies, can be downloaded from the project's Web site: http://www.fosd.net/fh.

We begin with a brief review of FSTCOMPOSER and proceed with a description of the overall FEATUREHOUSE architecture and how it integrates FSTCOMPOSER.

2.1 Representation and Composition

FSTCOMPOSER relies on a general model of the structure of software artifacts, called the *feature structure tree* (FST) model. An FST represents the essential structure of a software artifact and abstracts from language-specific details. For example, an artifact written in Java contains packages, classes, methods, and so forth, which are represented by nodes in its FST; a Haskell program contains equations, algebraic data types, type classes, etc., which contain further elements; a makefile or build script consists of definitions and rules that may be nested.

Each node of an FST has 1) a name that is the name of the corresponding structural element and 2) a type that represents the syntactic category of the corresponding structural element. For example, a Java class `Foo` is represented by a node `Foo` of type `Java class`. Essentially, an FST is a stripped-down abstract syntax tree: It contains only information that is necessary for the specification of the modular structure of an artifact and for its composition with other artifacts. The inner nodes of an FST denote modules (e.g., classes and packages) and the leaves carry the modules' content (e.g., method bodies and field initializers). We call the inner nodes *nonterminals* and the leaves *terminals*. For illustration, Fig. 1 depicts an excerpt of a class of a database system, taken from one of our case studies (Section 3.1). The complete class is located in a subpackage structure and contains 13 fields, 2 constructors, 58 methods, and 4 inner classes.

What code elements are represented as inner nodes and leaves? This depends on the language and on the level of *granularity* at which software artifacts are to be composed [31]. Different granularities are possible and might be desired in different contexts. For Java, we might represent only packages and classes but not methods or fields as FST nodes (a coarse granularity), or we might also represent statements or expressions as FST nodes (a fine granularity). In any case, the structural elements not represented in the FST are stored as text content of terminal nodes (e.g., the body of a method). In our experience, the granularity of Fig. 1 is usually appropriate for composition. We will return to the issue of granularity in Section 2.2.

1. Although AHEAD provides a language-independent *model* based on nested records, the support for different languages has been implemented for each language individually from scratch (see Section 6).

```
1  package com.sleepycat;
2  public class Database {
3    private DbState state;
4    private List triggerList;
5    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
6      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
7        for(int i=0; i<triggerList.size(); i+=1) {
8          DatabaseTrigger trigger = (DatabaseTrigger)triggerList.get(i);
9          trigger.databaseUpdated(this, locker, priKey, oldData, newData);
10       }
11   } // over 650 further lines of code...
12 }
```

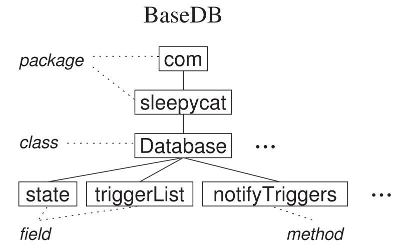
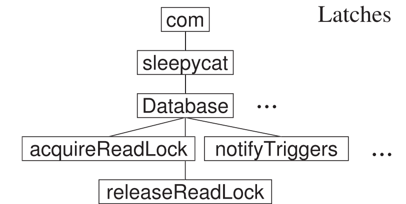


Fig. 1. Java code and FST of the artifact BaseDB, taken from the BERKELEY DB case study.

The composition of software artifacts proceeds by the superimposition of the corresponding FSTs, henceforth denoted by "•". Two FSTs are superimposed by merging their nodes, identified by their names, types, and relative positions, starting from the root and descending recursively. Fig. 2 illustrates the process of FST superimposition with the database example. The artifact BaseDB is superimposed with an artifact called Latches, of which again only a subset is shown. Their composition results in a class Database consisting of the union of the members of its instances in BaseDB and Latches. Basically, composing Latches with BaseDB adds two new methods acquireReadLock and releaseReadLock and extends method notifyTriggers of BaseDB via overriding (original defines how two method bodies are composed, which is similar to Java's super or AspectJ's proceed).

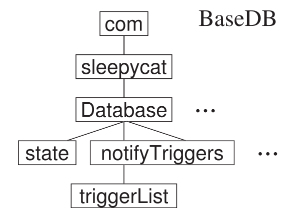
Generally, the composition of two leaves of an FST that contain further content (e.g., the two bodies of notifyTriggers) demands special treatment. The reason is that the content is not represented as a subtree but as plain text. Java method bodies are composed differently from fields, Haskell functions, or JavaCC grammar productions. Depending on the artifact language and node type, different rules for the composition of terminals apply. Often simple rules such as replacement, concatenation, specialization, or overriding suffice, but the approach is open to more sophisticated rules known from multidimensional separation of concerns [16] or software merging [32]. For example, in our case studies, we merge two method bodies via overriding, in which original defines how the bodies are merged. Note that original is not a new keyword added to the grammar of Java. We use a regular Java parser, which classifies original as a method

```
1  package com.sleepycat;
2  public class Database {
3    private void acquireReadLock() throws DatabaseException { ... }
4    private void releaseReadLock() throws DatabaseException { ... }
5    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
6      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
7        acquireReadLock();
8        original(locker,priKey,oldData,newData);
9        releaseReadLock();
10   } // 50 further lines of code...
11 }
```



•

```
1  package com.sleepycat;
2  public class Database {
3    private DbState state;
4    private List triggerList;
5    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
6      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
7        for(int i=0; i<triggerList.size(); i+=1) {
8          DatabaseTrigger trigger = (DatabaseTrigger)triggerList.get(i);
9          trigger.databaseUpdated(this, locker, priKey, oldData, newData);
10       }
11   } // over 650 further lines of code...
12 }
```



=

```
1  package com.sleepycat;
2  public class Database {
3    private DbState state;
4    private List triggerList;
5    private void acquireReadLock() throws DatabaseException { ... }
6    private void releaseReadLock() throws DatabaseException { ... }
7    protected void notifyTriggers(Locker locker, DatabaseEntry priKey,
8      DatabaseEntry oldData, DatabaseEntry newData) throws DatabaseException {
9        acquireReadLock();
10       for(int i=0; i<triggerList.size(); i+=1) {
11         DatabaseTrigger trigger = (DatabaseTrigger)triggerList.get(i);
12         trigger.databaseUpdated(this, locker, priKey, oldData, newData);
13       }
14       releaseReadLock();
15   } // over 700 further lines of code...
16 }
```

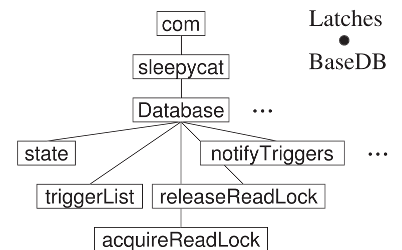


Fig. 2. Java code of Latches, BaseDB, and Latches • BaseDB.

name. FEATUREHOUSE searches for occurrences of original during the composition of two method bodies and substitutes each occurrence by the original method body (in the case of name clashes, with some semantics-preserving renaming). The only restriction is that original must not be used as a method name (or as any other kind of identifier). So, actually we have restricted the Java semantics minimally to support the composition of method bodies. As a result of the composition, we receive a syntactically correct Java program.

Technically, multiple software artifacts (e.g., code and corresponding documentation) can be aggregated in a *composition unit*. FSTCOMPOSER expects a list of units to be composed. The artifacts of a composition unit may be organized in a subdirectory structure. Without any further preparation, FSTCOMPOSER interprets subdirectories as nonterminals and the files located inside a subdirectory as terminals. Of course, if we intend to achieve a finer composition granularity than at the level of entire files (e.g., at the level of functions, classes and methods, and grammar rules, as in Fig. 2), we add further levels of nonterminals representing the artifacts' substructures, as we will explain next.

2.2 Generation and Automation

New languages can be easily plugged into FEATUREHOUSE. The idea is that, although artifact languages are very different, the process of software composition by superimposition is very similar. For example, the developers of AHEAD [6] and FeatureC++ [17] have extended the artifact languages Java and C++ by constructs (e.g., refines or Super) and mechanisms for composition. They have each implemented a parser, a superimposition algorithm, and a pretty printer[2]—all specific to the artifact language. We have introduced the FST model to be able to express superimposition independently of an artifact language [25]. Nevertheless, without automation, we had to provide, for each language (at the time, for Java and C#):

1. a parser and corresponding framework classes representing the parse tree,
2. an adapter that maps the parse tree to the FST,
3. language-specific composition rules (e.g., for merging method bodies), and
4. a pretty printer for writing superimposed FSTs to disk.

Overall, the process of implementing and integrating language support manually was time-consuming and error-prone. Usually, we (or our students) spent several weeks on making the parsers, adapters, and pretty printers work. Often, the initial versions of the manually implemented and integrated parsers, adapters, and pretty printers contained numerous bugs, so we had to spend significant time on debugging.

2.2.1 Generation

These problems motivated us to automate the integration of further languages and base it largely on the languages' grammars. This allows us to *generate* most of the code that must otherwise be provided and integrated manually (parser, adapter, pretty printer) and to experiment with different representations of software artifacts, as we will illustrate shortly. Our tool FSTGENERATOR expects the grammar of the language to be integrated in a specific format, called FEATUREBNF, and generates parser, adapter, and pretty printer accordingly. FEATUREBNF is similar to the Backus-Naur-Form but supports a number of extensions [33] and annotations, some of which are used by FSTGENERATOR, as we will explain. Using a grammar written in FEATUREBNF, FSTGENERATOR generates 1) an LL(k) parser that directly produces FST nodes and 2) a corresponding pretty printer. After the generation step, composition proceeds as follows: 1) the generated parser receives artifacts written in the target language and produces one FST per artifact, 2) FSTCOMPOSER performs the composition, and 3) the generated pretty printer writes the composed artifacts to disk. For the composition of the content of terminal nodes, we have developed and integrated a library of composition rules (e.g., rules for method overriding and for the concatenation of the statements of two constructors). Fig. 3 illustrates the interplay between FSTGENERATOR and FSTCOMPOSER; Table 1 lists the composition rules we have implemented so far (Section 3.1).

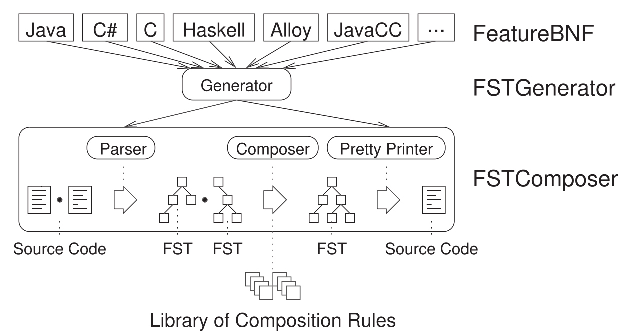


Fig. 3. The architecture of FEATUREHOUSE.

2.2.2 Attributes

To specify *how* artifacts of a language are represented as FSTs, programmers *annotate* the language's grammar with *attributes*. We explain the role of attributes using a simplified Java grammar. In Fig. 4, we depict an excerpt of the corresponding FEATUREBNF grammar that is relevant for classes and methods. For example, rule ClassDecl defines the structure of classes containing fields (VarDecl), constructors (ClassConstr), and methods (MethodDecl).

Without any attributes, FSTGENERATOR would create a single terminal node for each file; in our case, beside nonterminals denoting the enclosing directories and the enclosing Java file, there would only be a terminal node per class, and the class' members would appear as text in the terminal's content. Since this granularity is too coarse for our purposes (recall: we would like to exploit the internal structure of a Java file, as shown in Figs. 1 and 2), we use attributes to annotate the production rules that correspond to nonterminals (i.e., that contain further nodes).

Fig. 5 depicts an annotated version of our simple Java grammar. Attribute @FSTNonTerminal above rule ClassDecl states that classes are nonterminals that contain further

2. With "pretty printer" we refer to a tool, also known as unparser, that takes a parse tree or an FST and generates source code.

TABLE 1
Composition Rules Implemented in FEATUREHOUSE

Rule	Description
method overriding	merges two method bodies; original is used to inline one body into the other (one rule for Java and one for C#)
grammar-rule overriding	merges two grammar rules; original is used to inline the body of one rule into the body of the other
constructor concatenation	appends the statements of one constructor to the statements of the other
field specialization	assigns an initial value to a field in the case it did not have one before
implements-list union	takes the union of the types of two implements lists, excluding duplicates
modifier specialization	specializes modifiers similar to Java's subtyping rules
replacement	replaces one terminal node with the other
text-content concatenation	concatenates the text content of two terminal nodes

elements; all productions that follow and that are not annotated are automatically interpreted as terminals. It follows from the grammar that a node representing a class may have children representing its name, supertype, fields, constructors, and methods. The attribute's parameter, name, is used to assign the name of a class to the FST node representing the class.

With a single attribute, we have refined the composition granularity of Java artifacts. Now, Java FSTs have three levels (omitting packages and imports, for simplicity): 1) a root that represents the Java file, 2) a class that is a nonterminal, and 3) type names, methods, constructors, and fields that are terminals. Without the attribute, a Java FST consists only of a single node representing the corresponding Java file. In Fig. 6, we illustrate the difference between the two levels of granularity.

Besides @FSTNonTerminal, FSTGENERATOR supports several further kinds of attributes. For example, attribute @FSTTerminal is used to mark terminal nodes, as could be useful for representing classes as FST nodes. Although all production rules that are not annotated are interpreted as terminals or as text content of terminals, this attribute allows a programmer to define the name that appears in the FST node and the composition rule for merging the content of two corresponding terminals. For example, in Fig. 7 we

```
1  ClassDecl :
2    "class" Type "extends" ExtType "{"
3      (VarDecl)∗ (ClassConstr)∗ (MethodDecl)∗
4    "}";
5  VarDecl : Type ID ";";
6  MethodDeclaration :
7    Type ID "(" (FormalParamList)? ")" "{"
8      (Statement)∗
9    "}";
10 ...
```

Fig. 4. An excerpt of a simplified Java grammar.

```
1 @FSTNonTerminal(name="{Type}")
2 ClassDecl :
3   "class" Type "extends" ExtType "{"
4     (VarDecl)∗ (ClassConstr)∗ (MethodDecl)∗
5   "}";
6 ...
```

Fig. 5. An excerpt of a simplified Java grammar with annotations.

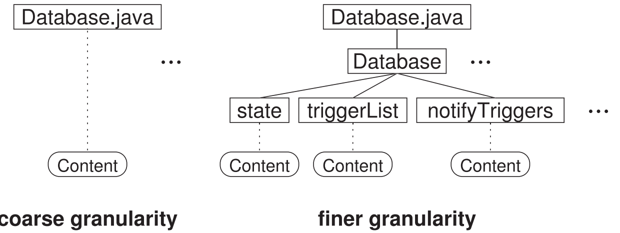


Fig. 6. Two granularities of representing Java artifacts as FSTs (controlled by a single attribute).

```
1 @FSTTerminal(name="{ID}({FormalParamList})",
2   compose="JavaMethodOverriding")
3 MethodDecl :
4   Type ID "(" ( FormalParamList )? ")" "{"
5     (Statement)∗
6   "}";
7 ...
```

Fig. 7. Annotating a method declaration with a name and a composition rule.

```
1  package composer.rules;
2
3  public class JavaMethodOverriding {
4    public final static String COMPOSITION_RULE_NAME =
5      "JavaMethodOverriding";
6    public static void compose(
7      FSTTerminal nodeA, FSTTerminal nodeB, FSTTerminal comp) {
8        ...
9    }
10 }
```

Fig. 8. Composition rule for Java method bodies.

specify that the name of a node representing a Java method receives the method's name (ID) followed by its formal parameters (FormalParamList). Production rules without explicitly assigned names receive proper default names.

Note that, without the possibility of specifying the name of an FST node, in many cases a superimposition would not be feasible. Recall that two nodes are superimposed if and only if their names (and types) are identical. For example, we can use parameter name to define that two classes are composed if their identifiers are identical (name="{Type}") or only if their identifiers *and* their supertypes are identical (name="{Type} {ExtType}").

Using parameter compose, we define which composition rule from the library is used when composing terminal nodes. In our simple Java grammar of Fig. 7, we define that methods are composed via method overriding. The value JavaMethodOverriding refers to an artifact-specific composition rule that is part of the library of composition rules.

In Fig. 8, we depict the interface of rule JavaMethodOverriding. A composition rule takes the terminal nodes to be composed as well as the result node as arguments. Then, it constructs the body of the result node such that it represents the composition of the two input nodes to be composed. This example illustrates the simplicity of writing composition rules. Of course, it is possible to write more

TABLE 2
Overview of the Languages Integrated in FEATUREHOUSE with FSTGENERATOR

	Java	C#	C	Haskell	Alloy	JavaCC
rules	135	229	45	78	56	170
nonterminals	10	17	2	13	11	16
terminals	13	18	9	9	11	16
attributes	42	53	21	24	27	61

sophisticated composition rules, for example, by parsing the individual bodies and performing syntax-tree transformations and type analysis.

3 Evaluation and Experience

To evaluate the generality, practicality, and scalability of our approach and to reveal open issues, we integrated six languages into FEATUREHOUSE and used it in the composition of 50 software systems of different sizes and domains. We report on key observations made and lessons learned.

3.1 Sample Languages and Systems

We integrated three imperative languages, two of them object-oriented (C, Java, and C#), one functional language (Haskell), one specification and modeling language (Alloy), and one domain-specific language for grammar specifications (JavaCC). In Table 2, we provide a summary of the overall number of grammar rules, the number of rules annotated to represent nonterminals, the number of rules annotated to represent terminals, and the overall number of attributes per language. FEATUREHOUSE, including all language plugins, is available on the project's Web site.

All of our 50 sample systems have been decomposed into composition units for different purposes, mostly by others. In our experiments, we composed them in different variants (reaching from one to hundreds). This was an incremental process during the last four years. The rationale of selecting such a large sample was to gain practical experience with FEATUREHOUSE and to address issues such as scalability and composition granularity. In Table 3, we summarize information on the sample software systems and their compositions. The source code of all software systems of our study is available on the FEATUREHOUSE Web site.

Note that some of the sample systems are related. BCJAK2JAVA, JAK2JAVA, JAMPACK, JRENAME, MIXIN, MMATRIX, and UNMIXIN belong to the AHEAD tool suite [34]. They share certain basic functionality such as parsers. Similarly, BALI2JAK, BALI2JAVACC, BALI2LAYER, and BALICOMPOSER belong to the BALI tool suite [34]. Furthermore, CHATSYSTEM comes in eight different variants. The variants have been developed independently in a course on software product-line engineering at the University of Magdeburg. The same applies to NOTEPAD, which comes in seven variants, independently developed in a course on feature-oriented design at The University of Texas at Austin. Next, we discuss our key observations.

3.2 Mandatory Properties

Our case studies demonstrate that FEATUREHOUSE is a very general tool and approach. We were able to integrate a number of different languages into FEATUREHOUSE, including languages of different programming paradigms as well as specification and domain-specific languages. Despite the considerable variety of the languages integrated, we identified a set of properties that are mandatory for a language to be plugged into FEATUREHOUSE:

1. The substructure of a software artifact must be a tree.
2. Every element of an artifact must provide a name that becomes the node's name and must belong to a syntactic category that becomes the node's type; an element must not contain two or more direct child elements with the same name and type.
3. Elements that do not have a hierarchical substructure represented in the FST (terminals) must come with composition rules in order to be composable.

3.3 Generality

Superimposition is only one of several composition approaches. It is especially useful in scenarios in which the code of components is available and their structures are compatible. Other scenarios such as black-box composition or the integration of structurally incompatible components are less suited for superimposition and should be handled by alternative composition approaches such as on-demand remodularization [7], [35] and component aggregation [36].

Our case studies have all been designed with superimposition in mind. Most of them are product lines whose features systematically refine the code of other features. Hence, we are not able to draw sound conclusions on the suitability of superimposition as compared to other composition techniques. This issue is outside the scope of this paper and is addressed partially by previous work [8], [37], [38]. Nevertheless, an extension of FEATUREHOUSE for quantification and weaving is motivated by this work and is discussed in Section 4.2.

3.4 Granularity

FEATUREHOUSE enables developers to adjust the granularity of composition by annotating the corresponding grammars. This way, we were able to experiment with different levels of granularity. For example, in Haskell, it was not clear which degree of granularity of superimposition is appropriate. It was clear that function definitions should be terminals, but not whether data type definitions should be terminals or nonterminals. After playing with some examples, we realized that it is quite useful to represent data types as nonterminals so that data type definitions can be extended by adding new type constructors. For example in Fig. 9, we compose two data-type declarations for the representation of binary operations by merging their corresponding type constructors (Sub for subtraction and Add addition; a deriving clause defines to which type class a data type belongs).

The flexibility in adjusting the composition granularity even makes it possible to compose unstructured text when it is viewed as a single terminal node. In this case, two text nodes could be merged via string concatenation. However, the more structure is exposed in an FST, the finer grained the composition can be (which typically makes the

TABLE 3
Overview of the Sample Systems

System	Domain	LOC	COMP	LANG
AJSTATS	Source code analysis tool	15 311	20	Java
ARITH	Arithmetic expression evaluator	442	15	Haskell
BALI2JAK	Grammar processing tool	13 527	11	Java
BALI2JAVACC	Grammar processing tool	14 139	11	Java
BALI2LAYER	Grammar processing tool	13 811	12	Java
BALICOMPOSER	Grammar processing tool	12 197	10	Java
BCJAK2JAVA	Code transformation tool	32 326	15	Java
BERKELEYDB	Embedded database engine	64 652	99	Java
CHATSYSTEM/BURKE	Network client and server	614	6	Java
CHATSYSTEM/DREILING	Network client and server	938	5	Java
CHATSYSTEM/BECKER	Network client and server	651	7	Java
CHATSYSTEM/WEISS	Network client and server	931	7	Java
CHATSYSTEM/SCHINK	Network client and server	873	7	Java
CHATSYSTEM/LUONG	Network client and server	862	9	Java
CHATSYSTEM/REHN	Network client and server	760	6	Java
CHATSYSTEM/THUEM	Network client and server	544	8	Java
CAN	Peer-to-peer protocol implementation	150	10	Alloy
EMAILSYSTEM	Simple email client	894	11	C
EPL	Arithmetic expression evaluator	99	12	Java
FFJ	Grammar of Feature Featherweight Java	289	2	JavaCC
FGL	Functional graph library	2 731	20	Haskell
GAMEOFLIFE	Computer game	1 656	14	Java
GPL	Graph library	2 439	26	Java
GPL	Graph library	2 148	26	C#
GRAPHLIB	Graph library	934	13	C
GRAPHMODEL	Model of a graph library	114	7	Alloy
GUIDSL	Product-line configuration tool	13 573	26	Java
JAK2JAVA	Code transformation tool	32 934	16	Java
JAMPACK	Code transformation tool	34 326	21	Java
JRENAME	Code transformation tool	31 120	17	Java
MIXIN	Code transformation tool	32 493	17	Java
MMATRIX	Code transformation tool	32 228	13	Java
MOBILEMEDIA8	Multimedia file manager	5 278	51	Java
NOTEPAD/QUARK	Text editor	1 397	10	Java
NOTEPAD/DELAWARE	Text editor	1 654	6	Java
NOTEPAD/WELLINGTON	Text editor	1 522	4	Java
NOTEPAD/SVETOSLAV	Text editor	1 627	6	Java
NOTEPAD/WEHRMAN	Text editor	1 716	5	Java
NOTEPAD/GUIMBARDA	Text editor	1 586	9	Java
NOTEPAD/ROBISON	Text editor	1 404	10	Java
PHONESYSTEM	Phone system model	96	4	Alloy
PKJAB	Chat network client	4 994	8	Java
PREVAYLER	In-memory database engine	6 867	6	Java
POSIXFILESYSTEM	Model of a POSIX file system	432	13	Alloy
RAROSCOPE	Compression library	428	5	Java
SUDOKU	Computer game	1 850	7	Java
TANKWAR	Computer game	3 184	15	Java
UNMIXIN	Code transformation tool	31 658	12	Java
VIOLET	UML model editor	9 789	88	Java
ZIPME	Compression library	5 479	35	Java

LOC: lines of code; COMP: composition units; LANG: artifact language

```
data BinOp = Sub deriving Eq;                •

data BinOp = Add deriving Show;              =

data BinOp = Add | Sub deriving Show, Eq;
```

Fig. 9. Composing data type declarations via superimposition, taken from the ARITH case study (no new composition rules are needed).

composition more expressive and easier to implement), but there is usually a limit. For example, representing and superimposing arithmetic expressions is certainly not useful because expressions do not have unique names and the order or their evaluation matters, as we explain next.

3.5 Uniqueness of Names

Unique names are central to composition with FEATUREHOUSE. Without unique names, superimposition does not work. For each sample language, we identified a proper level

of granularity at which elements have unique names and their composition is useful (i.e., there is enough structure for superimposition). In all languages, there are syntactic elements with unique names (in the scope of their parent elements), especially at coarse granularities (e.g., at the level of Java classes and Alloy signatures). However, as the granularity becomes finer, syntactic elements tend to have no or ambiguous names (e.g., at the level of Java statements or Alloy expressions). That is, the syntactic structure of the language affects the granularity at which artifacts can be meaningfully composed. Often, a fine granularity prohibits superimposition. There are two ways of attaining uniqueness of names: 1) adjust the attributes of the corresponding grammar to make composition coarser grained, and use a specific composition rule to compose the corresponding elements properly, as we have done, for example, for Java statements; 2) assign unique names manually, which may require changes to the language's syntax. Note that these approaches are related to each other: Refining the grain of composition often requires assigning proper names.

3.6 Element Order

An issue related to unique names is the order of elements. In our evaluation, we found that, typically, at a coarse granularity, the order of elements does not affect the program's or document's semantics (e.g., the order of Java methods or Alloy functions does not matter). As we made the granularity finer (by annotating the grammar), we observed that the elements' order becomes important in most languages (e.g., in the case of Java statements and Alloy expressions; in C, already at the level of functions). Superimposition is particularly useful at a level at which the elements' order may vary, which makes it is easy to add new elements (e.g., a new production rule to a JavaCC grammar). If the elements' order matters, it is difficult to insert elements between two existing elements (e.g., a statement in the middle of Java method). In this case, workarounds such as sandwiching are necessary.[3] When the elements' order matters, we can adjust the attributes of the corresponding grammar to make the composition coarser-grained and the elements' order immaterial, and use a specific composition rule to compose the corresponding elements properly. A classic example is the composition of two method bodies via a specific composition rule in which keyword `original` controls the order in which the statements of the bodies are merged.

3.7 Tradeoffs

Apparently, there is a tradeoff between granularity, expressiveness, and simplicity. At a fine granularity, we gain compositional expressiveness, but face problems regarding uniqueness of names and element ordering, which require complex and language-specific composition rules. At a coarse granularity, we lose compositional expressiveness but face fewer or no problems regarding names and order, and need less complex and fewer language-specific composition rules. For the languages we looked at, we found an acceptable balance between granularity, expressiveness, and simplicity because they provide a sufficient amount of structure.

3. Sandwiching is the process of dividing artifacts into parts to be able to add something in between [39].

TABLE 4
Amount of Boilerplate Code (LOC)

	Manual approach			Generative approach	
	Adapter	Pretty printer	Comp. rules	Comp. rules	Attributes
Java	1 366	424	214	178	42
C#	2 851	374	518	53*	53

* For C#, we could reuse most of the composition rules of Java.

3.8 Scalability

To learn about scalability, we included a number of software systems of substantial size in our sample. For example, we have composed 99 composition units of Berkeley DB with 64,000 lines of code. Our study shows that FEATUREHOUSE, even though it is an unoptimized prototype, scales well with the number of composition units and lines of code (24 seconds composition time). The numbers for other systems are similar or smaller. Note that composition granularity may influence composition time. In our case studies, a typical FST has only a low depth (3-10), which does not influence performance significantly.

3.9 Integration Effort

In earlier work, we developed a parser adapter (to bridge the gap between parse tree and FST) and a pretty printer manually for each language [25]; we call this the *manual approach*. With FEATUREHOUSE, we generate both automatically, based on attribute grammars; we call this the *generative approach*. In the manual approach, the effort of integrating a language was considerable—on the order of weeks. In contrast, in the generative approach, we usually spent only a few hours. For all languages, we were able either to rewrite an existing grammar given in EBNF, JavaCC, or ANTLR to the FEATUREBNF format (for Java, C#, Alloy, and JavaCC) or to write our own grammar based on the language specification (for C and Haskell). In Table 4, we list the amount of code we had to write in the manual and the generative approach for integrating Java and C#. We did not count generated code (in both approaches, we generated the parsers) and the code of the grammar specifications, which were publicly available. The generative approach reduces the need for code writing for Java to 11 percent and for C# to 3 percent of that in the manual approach.

In the manual approach, the granularity of composition is fixed. The adapter that translates a parse tree to an FST sets the granularity (i.e., decides which structural elements are represented as nonterminals and terminals) and is difficult and error-prone to write and change. In the generative approach, the attributes of the grammar define which structural elements are represented by nonterminals and terminals. Changing the attributes is a matter of minutes. This enabled us to experiment with different granularities.

Anyhow, if, for whatever reason, there is no grammar available for a particular language, the developer can resort to the manual approach and integrate an existing parser by means of an adapter. C++, with its inherently complex and ambiguous grammar, may be an example.

3.10 Composition Rules

Based on our experience, we have developed a number terminal composition rules and collected them in a library. These rules were reused in the integration of languages. Specifically, we have implemented composition rules for method overriding, grammar-rule overriding, constructor concatenation, field specialization, implements-list union, modifier specialization, replacement, and text-content concatenation (see Table 1). We can specify declaratively via grammar attributes for which kind of terminal we use which library composition rule, for example, compose=" JavaMethodOverriding" (see Section 2.2).

The interesting point is that we needed only a few rules, and the rules are very simple. The rule for method overriding is the most complex one. In Alloy—and later XML—we did not need specific composition rules at all. Superimposition and generic composition rules such as simple replacement sufficed in the case studies.

Another notable observation is that it was always sufficient to define a composition rule that is identical for all instances of a structural element of a language, for example, one rule that applies to all method bodies. Rules that differ from one element instance to another are possible but we refrained from implementing such rules as it would add considerable complexity and we did not encounter the need.

3.11 Tool Reliability

Finally, writing adapter code and a pretty printer is error-prone. After the manual integration of the Java parser into the initial version of FSTCOMPOSER, we detected lots of errors caused by bugs in the adapter code and in the pretty printer or by misconceptions regarding the role of some structural elements in the FST. For example, in the manual approach, we forgot to represent inner Java classes as nonterminals. In the generative approach, we stumbled over this issue early since it was exposed by the grammar because we annotated grammars top-down, starting from the root production and, at some point, reached inner classes and interfaces. Another example is the merge of initializer blocks of Java classes. In the manual integration, we simply did not think of this option until needed in BERKELEY DB. When annotating the grammar top-down, it became obvious.

3.12 Lessons Learned

Let us summarize the most significant observations we made and insights we gained during the integration of different languages and the composition of different software systems.

1. Superimposition is applicable to a wide range of code and noncode languages including object-oriented languages, functional languages, imperative languages, specification and modeling languages, and domain-specific languages.
2. Superimposition of FSTs scales to software projects of substantial size.
3. The time for preparing and annotating grammars is moderate compared to implementing adapters and pretty printers from scratch; varying the annotations varies the composition granularity and helps to cope with naming and ordering issues.
4. At fine granularities, elements often do not have unique identifiers (they are distinguished by the lexical order), which disallows superimposition. There are two solutions to this problem:
   a. adjust the attributes of the corresponding grammar to make composition coarser-grained, and use expressive but possibly language-dependent composition rules to compose the corresponding elements properly;
   b. assign unique names manually, which may require changes to the language's syntax.
5. The order of program elements and thus of FST nodes may matter. This makes superimposition difficult because, in some situations, workarounds such as sandwiching are necessary. Usually, this problem arises at a fine granularity. A solution is to adjust the attributes of the corresponding grammar to coarsen the granularity such that the parent node of the ordered elements becomes a terminal and then to compose the ordered elements properly by applying a composition rule.
6. In practice, only a few composition rules are needed; they can be reused by different languages and follow even fewer rule patterns.
7. The generative approach leads to more reliable tools than the manual approach. Developing a combination of a parser, adapter, and pretty printer from scratch is tedious and error-prone.

3.13 Threats to Validity

Note that our study was not designed to draw quantitative conclusions based on descriptive statistics, for example, regarding the frequency and feasibility of using superimposition. Rather, our goal was to demonstrate the practicality and generality of our approach and to gain insight into issues such as granularity and scalability. Nevertheless, we would like to comment on possible threats to validity in this context. As always, the selection of sample systems may threaten validity. Hence, we deliberately chose a large sample size, including systems of different domains and written in different languages. While we do not expect that further sample systems will change the big picture, we cannot generalize our findings to all kinds of languages. However, our experience suggests a catalog of mandatory properties that a language must have to be supported by FEATUREHOUSE (Section 3.2). A further threat to validity is that all sample systems have been developed with superimposition in mind, albeit mostly for different purposes. It would be interesting to see how FEATUREHOUSE performs in scenarios designed for other composition approaches such as aspect weaving, but this would require considerable effort to avoid bias and an entirely different study design to answer an entirely different set of research questions outside the scope of this paper.

4 EXTENSIONS

After the initial development of FEATUREHOUSE, the integration of a number of languages, and various case studies, we made a set of extensions and improvements to FEATUREHOUSE, which were motivated by new requirements and which diverge partly from the original approach

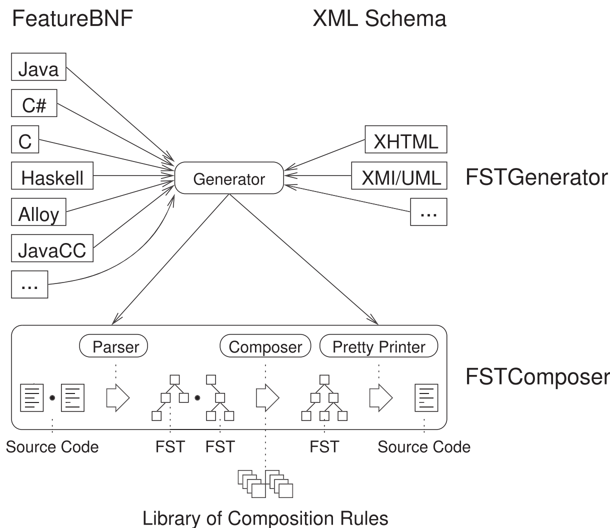


Fig. 10. The extended architecture of FEATUREHOUSE with support for XML-based languages.

and architecture. This is the reason why we discuss them separately, which also reflects the history of the FEATUREHOUSE project.

4.1 XML-Based Languages

Beside the six languages we had integrated into FEATUREHOUSE, we aimed at integrating XML and XML-based languages. XML is widely used to represent semistructured data, so supporting it would open FEATUREHOUSE for a whole new set of application scenarios. In a first attempt, we naively developed a grammar of XML in FEATUREBNF. We annotated the grammar such that XML elements were the nonterminal nodes and unstructured text content and element attributes were the terminal nodes. But we realized quickly that this approach would not succeed.

Integrating XML in general is not very interesting (and difficult due to missing names and relevant order). More interesting are XML-based languages, which are specified with XML Schema. XML Schema is used to define the grammar of an XML-based language. Popular examples are XHTML for Web sites and XMI for data exchange. To integrate a particular XML-based language into FEATUREHOUSE, we need its specific grammar (specifying which types of elements have which types of attributes and are allowed in which places, etc.)—using the plain XML grammar only is not sufficient because it is too unspecific.

There are two ways to integrate the grammar of an XML-based language, which we explain by means of XHTML. First, we can write a FEATUREBNF grammar that captures specifically the syntax of XHTML. This is a laborious task

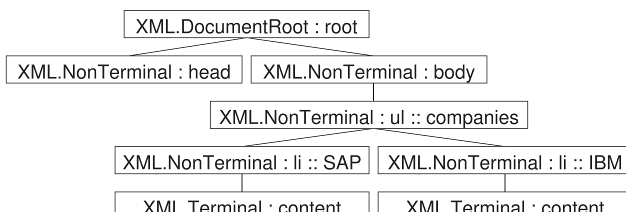


Fig. 11. An FST of an XHTML document.

```
1  <xs:element name="ul">
2    ...
3    <xs:complexType>
4      <xs:attributeGroup ref="attrs"/>
5      // mark unordered lists as nonterminals
6      <xs:attribute name="isTerminal" type="xs:boolean"
7        use="optional" default="false"/>
8      // support application−specific names via attribute fstname
9      <xs:attribute name="fstname" type="xs:string"/>
10   </xs:complexType>
11 </xs:element>
```

Fig. 12. Annotating XML schema to treat unordered lists as nonterminals with application-specific names.

since the lexical structure of XML-based languages is quite special, and the resulting grammar becomes huge and very complex. XML Schema has been invented for describing the structure of XML-based languages, and there are various industrial-strength tools available. So, actually, XML Schema should play the role of FEATUREBNF. Second, we could extend FEATUREHOUSE such that new languages are plugged in not only using FEATUREBNF but also using annotated XML Schema. To this end, we have to enrich XML Schema with the possibility of annotating it with the attributes defining terminals and nonterminals. This way, we can describe the syntax of XHTML quite easily and annotate it like FEATUREBNF-based languages.

In Fig. 10, we illustrate how we extended FEATUREHOUSE to support XML-based languages. Technically, we used a combination of XML attributes (to represent grammar annotations) and XSLT (to generate FSTs). In Fig. 11, we show an FST of a simple XHTML document that lists software companies; it consists of a head, a body, and an unnumbered list with two items. We define which elements are nonterminals and which are terminals (the corresponding XML elements are defined in an XML schema). Note that, while we use default names for the head and the body of an XHTML document (separated by ":"), we define additionally application-specific names for the list and the list items (separated by "::"). Technically, we define default names in the corresponding XML schema, as we illustrate in Fig. 12 for unordered lists, and we assign application-specific names via XSLT. Then, superimposition is performed as with any other language. We did not implement any composition rules specific to XML. Technical details are described elsewhere [30].

Based on the extended version of FEATUREHOUSE, we conducted a number of further case studies. In a first step, we integrated three languages: XHTML, XMI/UML,[4] and Ant for controlling build processes. Apart from the use of XML Schema, the integration is similar to that of FEATUREBNF-based languages. Note that, for Ant, we used a manual approach (cf. Section 3.9) as there is no static XML schema available.

Overall, we (de)composed six software systems, each of which is written in one of the three languages. Specifically, we decomposed the XHTML documentation of our GPL case study into features and composed them again depending on the feature selection. We decomposed the UML class, state, and sequence diagrams of the designs of

4. XMI is a general document format for data exchange. There is a special namespace for the representation of UML diagrams in XMI.

TABLE 5
Overview of the XML-Based Case Studies

System	Domain	LOC	COMP	LANG
GPLDOC	Documentation of GPL	484	16	XHTML
PHONE	Phone system	1 001	2	XMI/UML
ACS	Audio control system	2 080	4	XMI/UML
CMS	Conference management	2 077	9	XMI/UML
GBS	Gas boiler system	2 380	28	XMI/UML
BUILDER	Generic build script	57	6	Ant

LOC: lines of code; COMP: composition units; LANG: artifact language

a phone system, an audio control system, a conference management system, and a gas boiler system into features and composed them in different combinations generating different designs. Of course, we did not write XMI/UML code manually, but used ArgoUML[5] as a UML editor, which was only possible because we did not extend the syntax of XMI/UML—a strength of our approach. Finally, we decomposed an Ant build script into features and composed them such that the resulting build scripts control the build process according to the feature selection. Table 5 provides an overview of the XML-based case studies available on the FEATUREHOUSE Web site.

In summary, our findings were similar to those in the other case studies. Superimposition worked well at coarse and medium granularities at which XML elements have unique names and their order does not matter. At finer granularities, we had to assign names manually, which is especially easy in XML-based languages. For example, in the XHTML documentation of GPL, we assigned unique names to structural elements such as lists and list elements (using attribute fstname) to be able to extend them subsequently by other features.

A key insight is that our approach and architecture is general enough to integrate languages with a special syntax such as XML seamlessly. This makes it usable for a wide variety of composition problems outside conventional programming and specification languages.

4.2 Quantification and Weaving

Besides superimposition, other techniques for software composition have also been proposed, most notably composition by quantification and weaving [8], [29], [38], in which, when expressing changes, we specify declaratively the points at which the changes are applied. This idea has been explored in depth in work on multidimensional separation of concerns [7], aspect-oriented programming [40], adaptive programming [41], and strategic programming [42]. *Quantification* is the ability to apply the same generic change in multiple places [40]. In our approach, this means that we have to determine the locations in the FST where changes are to be applied. Applying the actual changes then corresponds to *weaving* in aspect orientation [40].

In the past, researchers identified complementary strengths and weaknesses of superimposition, on the one hand, and quantification and weaving on the other [8], [38], [43]. To integrate composition by quantification and weaving in our approach, we introduce the concept of a modification. A *modification* consists of two parts:

1. **Traversal specification.** A characterization of the FST nodes that will be affected during composition.
2. **Rewrite specification.** A specification of how these nodes will be affected.

A modification is performed by an FST traversal that determines the nodes to be modified and applies the necessary changes to them. A modification takes an FST as input and produces a modified FST as output. A traversal specification can yield a set of FST nodes as a result. This allows us to specify the modification of multiple nodes at once rather than each set member individually.

5. A graphical UML editor: http://argouml.tigris.org/.

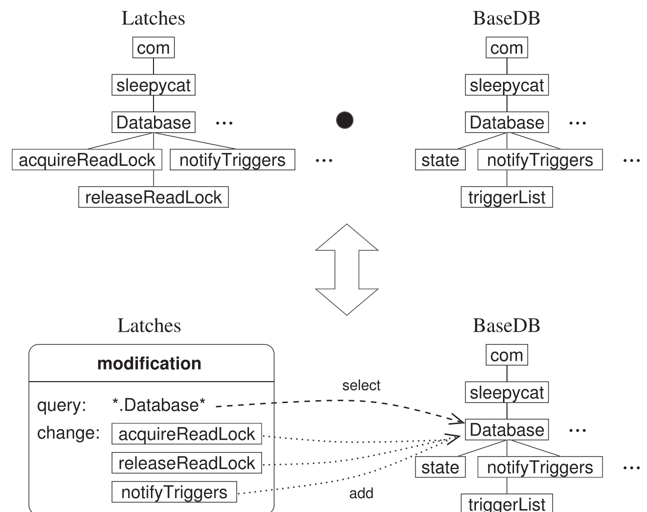


Fig. 13. Dual notions of composition: superimposition (top) and quantification and weaving (bottom).

An advantage of composition by quantification and weaving is that it enables us to address parts of a program more generically than superimposition, which always is applied at a root of an FST. That is, we can locate the places of change by a pattern on FST nodes that the structural elements of a program have to satisfy to be affected by a modification (e.g., "all methods in package util whose names begin with set"). For example, a feature could add a new field to *every* Java class of a package, regardless of the name of the class. Naturally, such a modification can be applied to a wide variety of programs. With superimposition, we have to specify each single target node, even though we change them all in the same way, which can result in considerable code replication [38]. Nevertheless, once the points of change are known, the two kinds of composition become equivalent. That is, once we have chosen a program, we can find an equivalent FST for every modification that, when superimposed with the program, produces the same results as applying the modification. Fig. 13 illustrates this similarity. In previous work, it has been shown that both kinds of composition have complementary strengths [38], so we include both of them in our model.

The extension of our FST model by support for quantification and weaving can be formulated again independently of a particular language. This illustrates that the FST model is very general. Another point is that, like

TABLE 6
Examples of Traversal Patterns along with Descriptions for Java FSTs

Traversal pattern	Description
com.sleepycat.Database : JavaClass	select class Database of package com.sleepycat
com.sleepycat.* : JavaClass	select every class of package com.sleepycat
com.sleepycat.* : *	select every element (class, method, etc.) of package com.sleepycat
com.* : JavaPackage + org.* : JavaPackage	select every subpackage of the packages com and org
com.* : JavaPackage - com.sleepycat.* : JavaPackage	select every subpackage of package com except the ones of com.sleepycat
com..state : JavaField	select every field state contained in any class of package com

with superimposition, the FST variant of quantification and weaving helps to condense the essence of related programming language mechanisms such as aspect weaving, which we discuss in Section 5.2 in the context of feature algebra. However, language independence always impairs expressiveness and so composition by quantification and weaving is not able to model all mechanisms of fully fledged languages such as AspectJ. We get back to this issue when describing one of our case studies later in this section.

We have implemented support for quantification and weaving on top of FEATUREHOUSE. To select a set of target nodes, a programmer can specify patterns such as "Database.* : JavaMethod" for selecting all methods of class Database. The pattern language is inspired by AspectJ but allows modifications to quantify also over syntactic categories (i.e., FST node types). In Table 6, we provide an overview of the most important constructs of the pattern language by means of examples. Technical details of the language are described elsewhere [29].

There are two types of rewrites: a rewrite that defines which new elements are added to the nodes selected by the corresponding traversal, and a rewrite that defines which new elements are composed with the selected nodes via terminal composition. In the current implementation, the nodes are given in text form (i.e., the code of the element to add or compose with).

Technically, traversal and rewrite specifications are embedded in an XML document, as illustrated in Fig. 14. Traversals and rewrites are straightforwardly implemented on top of FEATUREHOUSE's FST classes using visitors, pattern matching, and terminal composition rules.

```
1 <modification>
2   <type>introduction</type>
3   <traversal>..∗ : JavaClass</traversal>
4   <content>
5     <text>private Tracer t = new Tracer();</text>
6     <tType>Field</tType>
7   </content>
8 </modification>
```

Fig. 14. A modification introducing a field to every class.

TABLE 7
Overview of Case Studies on Quantification and Weaving

System	Domain	LOC	MOD	LANG
GPL	Graph library	2 439	12	Java
GPL	Graph library	2 148	12	C#
BERKELEY DB	Database system	58 030	4	Java
AJHOTDRAW	Graphics framework	43 368	181	Java

LOC: lines of code; MOD: modifications; LANG: artifact language

In three case studies, we implemented modifications for four software systems of different sizes written in two different languages. Table 7 lists relevant information on the case studies. The source code of all case studies is available on the FEATUREHOUSE Web site.

In a first study, we implemented several modifications that add new features to our GPL case study. To demonstrate the language independence of modifications, we implemented five new features with, in total, 12 modifications on top of the Java and a C# implementation of GPL (see Table 3). Although the quantification mechanism is language-independent, individual modifications are not. Hence, we created similar but different traversal and rewrite specifications in both languages.

In a second study, we implemented a generic tracing feature for BERKELEY DB. The tracing feature consists of four modifications. With over 300 classes, the four modifications affect large parts of the code base of BERKELEY DB. This indicates the high degree of genericity that can be achieved in the implementation of modifications as well as a certain scalability of our approach and tool. With superimposition, we would have to specify an FST consisting of extensions of all target classes.

In a third study, we explored to what extent it is possible to reimplement the aspects of AJHOTDRAW[6] with modifications. AJHOTDRAW is a Java/AspectJ framework for 2D graphics. Since modifications in FEATUREHOUSE support only a limited set of changes, we were not able to reimplement all of the 42 aspects. (Specifically, we were able to reimplement 23 aspects completely and 13 aspects partially. For example, the FST model does not capture dynamic crosscuts as implemented with cflow, which is due to the static nature of FSTs.)

Overall, the three case studies demonstrate that quantification and weaving are indeed language-independent mechanisms that can be implemented as part of FEATUREHOUSE. Although we did not aim at discussing the strengths and weaknesses of superimposition as compared to quantification and weaving, we observed that, especially for extensions that are identical for a number of program locations, modifications can decrease the amount of code replication considerably. We also found that, due to its syntactic nature, FST-based quantification and weaving is less expressive than control-flow-based mechanisms of aspect-oriented languages such as AspectJ. Apparently, there is a tradeoff between language independence and expressiveness. As control-flow-based language mechanisms are used rather infrequently [44], we decided in favor

6. http://sourceforge.net/projects/ajhotdraw.

of language independence, accepting the need for workarounds to mimic control-flow-based program extensions.

4.3 Summary

The two extensions of FEATUREHOUSE illustrate that the tool suite has considerable potential to be used in further application scenarios. Besides further composition operators and language families, we envision tools for visualization, documentation, and analysis, to name a few.

5 Perspectives

In this section, we present ongoing work that aims at improving software composition with FEATUREHOUSE. In particular, we discuss perspectives of the FEATUREHOUSE approach with regard to reliability, formalization, and automation based on our practical experience. Due to the lack of space, we must be more cursory here than in the previous sections.

5.1 Type System

An issue not addressed so far in FEATUREHOUSE is how we can assess correctness beyond syntactic and structural properties. As a challenge, we would like to guarantee that every composed system is well typed. In our database example, Latches refers to BaseDB. If the latter is not present in a composed system, Latches does not work properly and, due to dangling references, several type errors are reported for Latches at compile time. Specifically, there are two challenges of guaranteeing type correctness in FEATUREHOUSE. First, FEATUREHOUSE is language-independent. That is, we cannot use a type system tailored to a particular language. We need an at least partially language-independent solution. Second, in many projects there is variability in how to compose composition units. We may select specific combinations of composition units, for example, from optional and alternative units, and produce different variants of a product, as is common in software product lines [45]. Instead of type checking all possible combinations (an exponential number), we seek possibilities to type check individual composition units or closed sets of composition units in isolation.

With regard to the first challenge (language independence), we developed a formal calculus for software composition, called *g*DEEP [46]. The key idea of *g*DEEP is to model the structure of a composition unit as a nested record independently of a particular language, which is a representation equivalent to FSTs. Furthermore, *g*DEEP comes with a set of well-formedness and subtyping rules that define uniformly how units are composed type-safely. However, these language-independent rules alone cannot ensure type safety. They have to be complemented with language-specific type rules. In fact, there are two parts to type checking: the language-independent part checking compositions at the level of the module structure (nonterminal and terminal structure) and the language-dependent part checking terms written in specific artifact languages (content of terminals). We have shown that it is feasible to combine *g*DEEP with different languages such as Java and Haskell [46]. The distinction between a language-independent and a language-dependent part is analogous to the distinction between language-independent superimposition and language-dependent composition rules in FEATUREHOUSE,

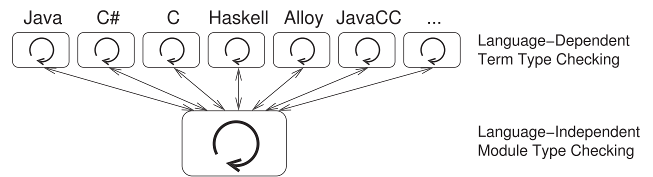


Fig. 15. The interplay between the language-independent and language-dependent parts of type checking in *g*DEEP.

as illustrated in Fig. 15. Hence, we are confident that a fully fledged type system based on *g*DEEP is not unrealistic.

With regard to the second challenge (type checking all composition units once, instead of all valid combinations), we developed a prototype of a language-independent checker, called FEATURETWEEZER, that takes a set of composition units and a description of their valid combinations (e.g., Latches requires BaseDB) and checks, without generating all valid products, whether there is any valid product that contains a type error [47]. Currently, we concentrate on a specific class of type errors: dangling references. The reason for this limitation is that a fully fledged, language-independent type system is not available. To this end, we have extended the FST model by cross-tree edges that represent references between program elements.

A reference is a very general concept. Modeling references as cross-tree edges is a common denominator for most languages. For example, a Java method invocation (i.e., the language-independent part of method lookup) is represented by an edge from the caller to the callee; a JavaCC reference is represented by an edge from a nonterminal to the corresponding production rule. Based on this representation, FEATURETWEEZER checks a set of composition units once and guarantees, given that the check has been successful, that no valid combination contains a dangling reference. This saves especially much time if there are many valid combinations, such as in software product lines, and checking all of them individually is infeasible. The algorithm is a straightforward extension of previous work on type checking product lines [48], [49], [50], [51], adapted to a language-independent model. In a nutshell, FEATURETWEEZER checks, for every reference, whether the target node (e.g., the called method) is present in every valid product in which also the source node (the calling method) is present. To this end, FEATURETWEEZER uses propositional logic and SAT solver technology to check references. We envision to extend FEATURETWEEZER to a fully fledged type system based on the insight we gained with *g*DEEP.

5.2 Feature Algebra

The *g*DEEP calculus is a first step toward formalizing and reasoning about feature composition. In a parallel line of work, we have been exploring how feature composition can be formalized using algebra [24]. A motivation of this work is to raise the abstraction level further, from logic (as in *g*DEEP) to simple algebraic structures such as semigroups and semimodules. Again, the FST model is the foundation. To enable formal reasoning, the feature algebra describes features as algebraic terms and feature composition as a fundamental operation. Actually, there are several atomic operations that resemble different composition mechanisms, including superimposition and quantification and weaving. For example, feature BaseDB of Fig. 1 is represented as a sum (denoted with $\oplus$) of introductions:

$$\begin{aligned}\text{BaseDB} = &\ com.sleepycat.Database.state\\ &\oplus com.sleepycat.Database.triggerList\\ &\oplus com.sleepycat.Database.notifyTriggers\\ &\oplus \dots.\end{aligned}$$

Each introduction represents a path in the FST beginning from the root (prefixes represent paths; FST node types are not shown here for brevity). Superimposition is represented by adding new introductions to a given introduction sum. So, composing feature Latches with feature BasedDB is achieved by the sum of their introductions.

Quantification and weaving is represented by the operation of modification application (denoted with $\odot$). A modification m is applied to a sum $i_1 \oplus \cdots \oplus i_n$ of introductions by applying it to all of its summands, which is captured by a distributivity law:

$$m \odot (i_1 \oplus \cdots \oplus i_n) = (m \odot i_1) \oplus \cdots \oplus (m \odot i_n).$$

Feature algebra is a means of exploring the design decisions we implemented in FEATUREHOUSE formally. Interestingly, we came to very similar conclusions as with our work on FEATUREHOUSE, especially with regard to the properties that are mandatory to make a language feature-ready (elements must have unique names, etc.). The feature algebra helps to analyze the effects of certain design decisions formally. For example, under which circumstances is feature composition associative, idempotent, or even commutative? We gained insight that was difficult to gain with tools like FEATUREHOUSE, which contains a lot of distracting functionality. For example, we found that composition by quantification and weaving results in a reduction of compositional flexibility compared to superimposition, which is counterintuitive since changes can be expressed more generically and declaratively than with superimposition. Supporting the full power of quantification (each composition unit may affect every other unit) leads to a composition operator that is not even associative. (We consider associativity an important property of composition.) Also, the algebra helped us to compare different composition tools such as AspectJ, CaesarJ, and FeatureC++ [24]. For example, we found that, in principle, CaesarJ and FeatureC++ are similar, whereas AspectJ is different as it does not support (symmetric) superimposition. Feature algebra provides a means of extracting the essence of the difference.

In the long term, we intend to use the feature algebra not only to describe the mechanisms implemented in FEATUREHOUSE, but also as intermediate language to specify the behavior of our composition tool. Software components are translated to algebraic expressions that can be manipulated based on algebraic laws. Algebraic optimization facilitates design optimization. That is, by manipulating feature-algebraic terms we manipulate programs and their designs. This approach is called *architectural metaprogramming* [52] because it lifts metaprogramming (programs manipulate programs) to the architectural level (programs manipulate algebraic expressions that represent programs). Our work on feature algebra and FEATUREHOUSE is an important step toward realizing the vision of architectural metaprogramming.

6 RELATED WORK

Although manifested and implemented differently, several languages provide support for superimposition of different kinds of artifacts, for example, Jiazzi [11], Classbox/J [14], Hyper/J [16], and Jak [6] for Java, FeatureC++ [17] for C++, Xak [19] for XML, and others [6], [9], [20], [21], [22], [53]. In turn, superimposition is based on a large corpus of work on extending objects and classes noninvasively, for example, mixins [54], traits [55], virtual classes [56], and object composition [57]. Work on superimposition builds on these approaches, generalizes them to different kinds of software artifacts, and applies extensions consistently to entire sets of entities.

While it has been noted that there is a unique core common to all composition mechanisms based on superimposition [6], researchers have not condensed the essence of superimposition into a general methodology and tool chain for software composition. A notable exception is the work of Batory et al. who, for the first time, stressed the language-independent nature of software composition by superimposition [6]. Batory et al. have proposed the AHEAD model for superimposition, based on nested records, that was a starting point for our work. We have adapted and evolved the model toward our FST model. In contrast to AHEAD's nested records, the FST model distinguishes between terminals and nonterminals and presume a fixed order of elements. This and the tree structure place the FST model closer to the implementation level and allowed us to directly derive an implementation, which has not been done by AHEAD (each AHEAD tool for each language has been developed from scratch). So, we believe that our FST model captures the essence of superimposition more precisely. It is language-independent and automates the integration of new languages. We envision further algorithms to be integrated in FEATUREHOUSE that operate on FSTs and their algebraic representations to compose, visualize, optimize, and verify software. Thus, FEATUREHOUSE provides a general framework not only for different languages but also for different algorithms that aim at reasoning about software composition language-independently.

In the context of our FST model, quantification is modeled as a tree walk in which each node is visited and a predicate specifies whether the node is modified or not. Harrison et al. [58] propose a sophisticated set of rewriting rules that are based on tree walks. Much like in our earlier work (manual approach; cf. Section 3.9) [25], they define a general, language-independent composition algorithm that parses and prints the code of artifacts by calling language-specific plug-ins, which are analogous to the adapters and pretty printers in the manual approach. They applied their approach to Java (source and binary) and Ant.

Recent work in model composition [59], [60], [61], [62] aims at developing a general framework for composing different kinds of models. Our approach can be applied to models (see Section 4.1), but aims also at nonmodeling languages. Like FEATUREHOUSE, Hyper/J, CaesarJ, Xak, and FeatureC++ support the combination of superimposition and quantification, but not independent of the language.

Software composition is related to the broad field of software merging, whose goal is to merge different versions

of a software system not only at the module level but at all levels of granularity by using syntactic, semantic, and evolutionary information [32]. Especially for the implementation of artifact-specific composition rules, superimposition can benefit from these developments.

In a parallel line of research, we have implemented a product-line tool, called CIDE, that allows a developer to decompose legacy software into a product line, type check *all* products of a product line, and visualize and resolve feature interactions [31], [51]. CIDE also pursues a generative approach of integrating new languages [63] based on the same grammar format as in FEATUREHOUSE but using different attributes; initially, FEATUREBNF was developed for CIDE. CIDE uses the entire parse tree; thus, it does not require a mapping to terminals and nonterminals of an FST. The coordinated development of FEATUREHOUSE and CIDE allows us to use grammars in both projects. CIDE has been used to decompose some of our case studies into superimposable units [64].

Delta-oriented programming is partially based on superimposition [65]. A *delta* encapsulates all changes that a feature makes to a program. In contrast to our composition units, deltas can even specify the removal of elements of a given program. Hence, delta-oriented programming diverges from the concept of superimposition and also of quantification and weaving. Recently, it has been shown that the feature algebra can be embedded in an algebra that describes delta composition [66]. It would be interesting to explore whether and how FEATUREHOUSE could be extended to support delta composition. For example, it seems quite straightforward to allow programmers to remove nodes from an FST, which is a basic delta operation.

Azanza et al. present a general approach to the incremental development of model-based software [62]. They describe transformations as model deltas that, when composed, deliver a complete model. They establish a relationship between a metamodel and its corresponding delta metamodel, show how model deltas can be defined as model changes (additions), explain how deltas can be composed using domain-specific composition algorithms, and propose metamodel annotations to specify these algorithms. On the one hand, the key innovation is that the approach of Azanza et al. allows programmers to plug in new composition algorithms. On the other hand, the authors do not focus on language independence in the sense of FEATUREHOUSE.

Finally, we extended FEATUREHOUSE to also support, besides composition via superimposition and quantification and weaving, software merging in revision control systems [67]. While this extension diverged in certain aspects substantially from the FEATUREHOUSE approach, we were able to build a merge engine on the FST model. This facilitated an easy integration of support for multiple languages (Java, C#, and Python) as well as the enrichment of the merge process with information (via grammar attributes) on how certain code fragments are merged.

7 Conclusion

We provided a coherent and complete view of our experience and insight gained with FEATUREHOUSE. This paper subsumes and extends our previous work on language-independent, automated software composition [27], [28], [29], [30]. Compared to previous work, we report on experience in a substantial number of case studies, a diverse selection of languages, a number of extensions of our approach, and perspectives that arise from our holistic and retrospective view on the FEATUREHOUSE approach.

FEATUREHOUSE is an approach and a set of accompanying tools for language-independent software composition. In FEATUREHOUSE, we model software artifacts by tree structures and composition by tree superimposition as well as tree traversals and rewrites. The FST model abstracts from the specifics of a particular programming language or tool. Any reasonably structured software artifact that can be represented as an FST can be composed by our approach.

FSTGENERATOR generates, on the basis of an attribute grammar, an FST representation and a pretty printer for a given language. FSTCOMPOSER composes FSTs generically via both superimposition and quantification and weaving. From the integration of various languages (including programming, specification and modeling, and domain-specific languages) and the application to several programs of different sizes, written in different languages, we learned much about our approach and the properties and problems of languages to be integrated. In particular, we found that the composition granularity is influenced by the language's syntactic structure. A fine granularity is expressive but may be infeasible because of naming and ordering issues. Composing artifacts at a coarser grain alleviates these problems but decreases expressiveness. The generative approach of FEATUREHOUSE allows developers to easily adjust the granularity to find a proper balance per language.

In the future, we would like to extend FEATUREHOUSE by further languages, composition operators, a type system, and tools, for example, for visualization, documentation, and analysis. Our work on feature algebra and on language-independent type checking is a foundation for this endeavor.

Acknowledgments

The authors are grateful to Don Batory for helpful comments on earlier drafts of this paper, Sebastian Scharinger and Alexander von Rhein for implementing the Java and C# parsers of FSTCOMPOSER, Stefan Boxleitner for integrating quantification and weaving, Jens Dörre for adding XML-based languages, Marko Rosenmüller and Norbert Siegmund for their support in developing the C grammar, Wolfgang Scholz for implementing FEATURETWEEZER, and the anonymous reviewers of this article and earlier conference versions. Apel's and Lengauer's work is supported by the German Research Foundation (DFG—AP 206/2, AP 206/4, and LE 912/13). Kästner's work was supported by the European Research Council (ERC #203099) while he was working as a postdoctoral researcher in the Programming Language Group of Klaus Ostermann at Philipps University Marburg, Germany.

References

[1] H. Ossher and W. Harrison, "Combination of Inheritance Hierarchies," *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pp. 25-40, 1992.

[2] L. Bouge and N. Francez, "A Compositional Approach to Superimposition," *Proc. 15th Int'l Symp. Principles of Programming Languages,* pp. 240-249, 1988.

[3] S. Katz, "A Superimposition Control Construct for Distributed Systems," *ACM Trans. Programming Languages and Systems,* vol. 15, no. 2, pp. 337-356, 1993.

[4] Y. Smaragdakis and D. Batory, "Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 2, pp. 215-255, 2002.

[5] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," *Proc. European Conf. Object-Oriented Programming,* pp. 419-443, 1997.

[6] D. Batory, J. Sarvela, and A. Rauschmayer, "Scaling Step-Wise Refinement," *IEEE Trans. Software Eng.,* vol. 30, no. 6, pp. 355-371, June 2004.

[7] P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr., "N Degrees of Separation: Multi-Dimensional Separation of Concerns," *Proc. 21st Int'l Conf. Software Eng.,* pp. 107-119, 1999.

[8] M. Mezini and K. Ostermann, "Variability Management with Feature-Oriented Programming and Aspects," *Proc. 12th Int'l Symp. Foundations of Software Eng.,* pp. 127-136, 2004.

[9] M. Sihman and S. Katz, "Superimpositions and Aspect-Oriented Programming," *Computer J.,* vol. 46, no. 5, pp. 529-541, 2003.

[10] J. Bosch, "Super-Imposition: A Component Adaptation Technique," *Information and Software Technology,* vol. 41, no. 5, pp. 257-273, 1999.

[11] S. McDirmid and W. Hsieh, "Aspect-Oriented Programming with Jiazzi," *Proc. Second Int'l Conf. Aspect-Oriented Software Development,* pp. 70-79, 2003.

[12] N. Nystrom, S. Chong, and A. Myers, "Scalable Extensibility via Nested Inheritance," *Proc. 19th Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 99-115, 2004.

[13] M. Odersky and M. Zenger, "Scalable Component Abstractions," *Proc. 20th Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 41-57, 2005.

[14] A. Bergel, S. Ducasse, and O. Nierstrasz, "Classbox/J: Controlling the Scope of Change in Java," *Proc. 20th Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 177-189, 2005.

[15] D. Hutchins, "Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes," *Proc. 21st Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 1-19, 2006.

[16] H. Ossher and P. Tarr, "Hyper/J: Multi-Dimensional Separation of Concerns for Java," *Proc. 22nd Int'l Conf. Software Eng.,* pp. 734-737, 2000.

[17] S. Apel, T. Leich, M. Rosenmüller, and G. Saake, "FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming," *Proc. Fourth Int'l Conf. Generative Programming and Component Eng.,* pp. 125-140, 2005.

[18] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich, "Access Control in Feature-Oriented Programming," *Science of Computer Programming,* special issue on feature-oriented software development, vol. 77, no. 3, pp. 174-187, 2012.

[19] F. Anfurrutia, O. Díaz, and S. Trujillo, "On Refining XML Artifacts," *Proc. Seventh Int'l Conf. Web Eng.,* pp. 473-478, 2007.

[20] S. Clarke, W. Harrison, H. Ossher, and P. Tarr, "Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code," *Proc. 14th Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 325-339, 1999.

[21] K. Czarnecki and M. Antkiewicz, "Mapping Features to Models: A Template Approach Based on Superimposed Variants," *Proc. Fourth Int'l Conf. Generative Programming and Component Eng.,* pp. 422-437, 2005.

[22] T. Kamina and T. Tamai, "Lightweight Scalable Components," *Proc. Sixth Int'l Conf. Generative Programming and Component Eng.,* pp. 145-154, 2007.

[23] G. Freeman, D. Batory, and G. Lavender, "Lifting Transformational Models of Product Lines: A Case Study," *Proc. First Int'l Conf. Model Transformation,* pp. 16-30, 2008.

[24] S. Apel, C. Lengauer, B. Möller, and C. Kästner, "An Algebraic Foundation for Automatic Feature-Based Program Synthesis," *Science of Computer Programming,* vol. 75, no. 11, pp. 1022-1047, 2010.

[25] S. Apel and C. Lengauer, "Superimposition: A Language-Independent Approach to Software Composition," *Proc. Seventh Int'l Symp. Software Composition,* pp. 20-35, 2008.

[26] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann, "An Overview of CaesarJ," *Trans. Aspect-Oriented Software Development I,* vol. 3880, pp. 135-173, 2006.

[27] S. Apel, C. Kästner, and C. Lengauer, "FeatureHouse: Language-Independent, Automated Software Composition," *Proc 31st. IEEE Int'l Conf. Software Eng.,* pp. 221-231, 2009.

[28] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Feature (De)Composition in Functional Programming," *Proc. Eight Int'l Conf. Software Composition,* pp. 9-26, 2009.

[29] S. Boxleitner, S. Apel, and C. Kästner, "Language-Independent Quantification and Weaving for Feature Composition," *Proc. Eight Int'l Conf. Software Composition,* pp. 45-54, 2009.

[30] J. Dörre, "Feature-Oriented Composition of XML Artifacts," master's thesis, Dept. of Informatics and Math., Univ. of Passau, 2009.

[31] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in Software Product Lines," *Proc. 30th Int'l Conf. Software Eng.,* pp. 311-320, 2008.

[32] T. Mens, "A State-of-the-Art Survey on Software Merging," *IEEE Trans. Software Eng.,* vol. 28, no. 5, pp. 449-462, May 2002.

[33] D. Wile, "Abstract Syntax from Concrete Syntax," *Proc. 19th Int'l Conf. Software Eng.,* pp. 472-480, 1997.

[34] D. Batory, J. Liu, and J. Sarvela, "Refinements and Multi-Dimensional Separation of Concerns," *Proc. Ninth European Software Eng. Conf. and the Int'l Symp. Foundations of Software Eng.,* pp. 48-57, 2003.

[35] M. Mezini and K. Ostermann, "Integrating Independent Components with On-Demand Remodularization," *Proc. 17th Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 52-67, 2002.

[36] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming.* Addision-Wesley, 2002.

[37] R. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating Support for Features in Advanced Modularization Technologies," *Proc. 19th European Conf. Object-Oriented Programming,* pp. 169-194, 2005.

[38] S. Apel, T. Leich, and G. Saake, "Aspectual Feature Modules," *IEEE Trans. Software Eng.,* vol. 34, no. 2, pp. 162-180, Mar./Apr. 2008.

[39] D. Parnas, "Designing Software for Ease of Extension and Contraction," *Proc. Third Int'l Conf. Software Eng.,* pp. 264-277, 1978.

[40] H. Masuhara and G. Kiczales, "Modeling Crosscutting in Aspect-Oriented Mechanisms," *Proc. European Conf. Object-Oriented Programming,* pp. 2-28, 2003.

[41] K. Lieberherr, B. Patt-Shamir, and D. Orleans, "Traversals of Object Structures: Specification and Efficient Implementation," *ACM Trans. Programming Languages and Systems,* vol. 26, no. 2, pp. 370-412, 2004.

[42] R. Lämmel, E. Visser, and J. Visser, "Strategic Programming Meets Adaptive Programming," *Proc. Second Int'l Conf. Aspect-Oriented Software Development,* pp. 168-177, 2003.

[43] A. Colyer and A. Clement, "Large-Scale AOSD for Middleware," *Proc. Third Int'l Conf. Aspect-Oriented Software Development,* pp. 56-65, 2004.

[44] S. Apel, "How AspectJ is Used: An Analysis of Eleven AspectJ Programs," *J. Object Technology,* vol. 9, no. 1, pp. 117-142, 2010.

[45] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns.* Addison-Wesley, 2002.

[46] S. Apel and D. Hutchins, "A Calculus for Uniform Feature Composition," *ACM Trans. Programming Languages and Systems,* vol. 32, no. 5, article 19, pp. 1-33, 2010.

[47] S. Apel, W. Scholz, C. Lengauer, and C. Kästner, "Language-Independent Reference Checking in Software Product Lines," *Proc. Second Int'l Workshop Feature-Oriented Software Development,* pp. 65-71, 2010.

[48] S. Thaker, D. Batory, D. Kitchin, and W. Cook, "Safe Composition of Product Lines," *Proc. Sixth Int'l Conf. Generative Programming and Component Eng.,* pp. 95-104, 2007.

[49] B. Delaware, W. Cook, and D. Batory, "Fitting the Pieces Together: A Machine-Checked Model of Safe Composition," *Proc. Seventh European Software Eng. Conf. and the Int'l Symp. Foundations of Software ng.,* pp. 243-252, 2009.

[50] S. Apel, C. Kästner, A. Größlinger, and C. Lengauer, "Type Safety for Feature-Oriented Product Lines," *Automated Software Eng.,* vol. 17, no. 3, pp. 251-300, 2010.

[51] C. Kästner, S. Apel, T. Thüm, and G. Saake, “Type Checking Annotation-Based Product Lines,” *ACM Trans. Software Eng. and Methodology,* vol. 31, no. 3, article 14, pp. 1-39, 2012.
[52] D. Batory, “Program Refactorings, Program Synthesis, and Model-Driven Design (Keynote),” *Proc. Int'l Conf. Compiler Construction,* pp. 156-171, 2007.
[53] L. Bergmans and M. Aksit, “Composing Crosscutting Concerns Using Composition Filters,” *Comm. ACM,* vol. 44, no. 10, pp. 51-57, 2001.
[54] M. Flatt, S. Krishnamurthi, and M. Felleisen, “Classes and Mixins,” *Proc. 25th Int'l Symp. Principles of Programming Languages,* pp. 171-183, 1998.
[55] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black, “Traits: A Mechanism for Fine-Grained Reuse,” *ACM Trans. Programming Languages and Systems,* vol. 28, no. 2, pp. 331-388, 2006.
[56] O. Madsen and B. Moller-Pedersen, “Virtual Classes: A Powerful Mechanism in Object-Oriented Programming,” *Proc. Int'l Conf. Object-Oriented Programming, Systems, Languages, and Applications,* pp. 397-406, 1989.
[57] L. Bettini, V. Bono, and B. Venneri, “Delegation by Object Composition,” *Science of Computer Programming,* vol. 76, no. 11, pp. 992-1014, 2011.
[58] W. Harrison, H. Ossher, and P. Tarr, “General Composition of Software Artifacts,” *Proc. Fifth Int'l Symp. Software Composition,* pp. 194-210, 2006.
[59] P. Bernstein, A. Halevy, and R. Pottinger, “A Vision for Management of Complex Models,” *SIGMOD Record,* vol. 29, no. 4, pp. 55-63, 2000.
[60] D. Kolovos, R. Paige, and F. Polack, “Merging Models with the Epsilon Merging Language (EML),” *Proc. Ninth Int'l Conf. Model-Driven Eng., Languages, and Systems,* pp. 215-229, 2006.
[61] J. Dingel, Z. Diskin, and A. Zito, “Understanding and Improving UML Package Merge,” *Software and Systems Modeling,* vol. 7, no. 4, pp. 443-467, 2008.
[62] M. Azanza, D. Batory, O. Diaz, and S. Trujillo, “Domain-Specific Composition of Model Deltas,” *Proc. Third Int'l Conf. Model Transformation,* pp. 16-30, 2010.
[63] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory, “Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach,” *Proc. Int'l Conf. Objects, Models, Components, Patterns,* pp. 174-194, 2009.
[64] C. Kästner, S. Apel, and M. Kuhlemann, “A Model of Refactoring Physically and Virtually Separated Features,” *Proc. Eight Int'l Conf. Generative Programming and Component Eng.,* pp. 157-166, 2009.
[65] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella, “Delta-Oriented Programming of Software Product Lines,” *Proc. 14th Int'l Software Product Line Conf.,* pp. 77-91, 2010.
[66] D. Clarke, M. Helvensteijn, and I. Schaefer, “Abstract Delta Modeling,” *Proc. Ninth Int'l Conf. Generative Programming and Component Eng.,* pp. 13-22, 2010.
[67] S. Apel, J. Liebig, B. Brandl, C. Lengauer, and C. Kästner, “Semistructured Merge: Rethinking Merge in Revision Control Systems,” *Proc. 19th European Software Eng. Conf. and the Int'l Symp. Foundations of Software Eng.,* pp. 190-200, 2011.

Sven Apel received the PhD degree in computer science from the University of Magdeburg, Germany, in 2007. He is the leader of the Software Product-Line Group funded by the esteemed Emmy Noether Programme of the German Research Foundation (DFG). The group resides at the University of Passau, Germany. His research interests include novel programming paradigms, software engineering and product lines, and formal and empirical methods. He is the author or coauthor of more than 100 peer-reviewed scientific publications. Since 2009 he has been a member of the IFIP Working Group 2.11 (Program Generation), and he has been a program committee member for several highly ranked International conferences. His work has received awards from the Ernst–Denert Foundation and the Karin–Witte Foundation.

Christian Kästner received the PhD degree in computer science from the University of Magdeburg, Germany, in 2010. He is an assistant professor at the Institute for Software Research at the Carnegie Mellon University, Pittsburgh, Pennsylvania. His research interests include correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, empirical evaluations, and refactoring. For his dissertation on virtual separation of concerns, he received the prestigious GI-Dissertation Award.

Christian Lengauer occupies the Chair for Programming at the University of Passau. Previous appointments were at the University of Texas at Austin and the University of Edinburgh. His research interests include parallel programming, notably loop parallelization, and programming paradigms and methods, among them feature orientation. He is the chair of the steering committee of the annual conference series Euro-Par and of the IFIP Working Group 2.11 on program generation.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.