FEATUREHOUSE: Language-Independent, Automated Software Composition

Sven Apel
Department of Informatics and Mathematics
University of Passau
apel@uni-passau.de

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

Christian Lengauer
Department of Informatics and Mathematics
University of Passau
lengauer@uni-passau.de

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, JavaCC, and XML. Several case studies demonstrate the practicality and scalability of our approach and reveal insights into the properties a language must have in order to be ready for superimposition.*

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, e.g., 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 multi-team software development [33], the extension of distributed programs [13], the implementation of collaboration-based designs [38], feature-oriented programming [8, 36], multi-dimensional separation of concerns [39], aspect-oriented programming [28,30], and software component adaptation [12]. Although very different, all these applications pursue superimposition of hierarchically organized program constructs on the basis of their nominal and structural similarities.

It has been noted that, when composing software, not only code artifacts – possibly written in different programming languages – have to be considered but also non-code artifacts, e.g., models, documentation, grammar files, or makefiles [8]. Thus, as a composition technique, superimposition should be applicable to a wide range of software artifacts. While there are various tools that support superimposition for code artifacts [4, 8, 9, 20, 28, 31, 32, 34, 37] and non-code artifacts [1, 8, 14, 16, 18, 21], they appear all different, are dedicated to and embedded differently in their respective host languages, and their implementation and integration requires a major effort. A reason is that, 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 general approach to the composition of software artifacts written in different languages and offer a supporting framework and tool chain, called FEATUREHOUSE. FEATUREHOUSE is a descendant of Batory's AHEAD program generator [8] and builds on our previous work on language-independent software representation [6] and composition [5], as we will explain.

In a nutshell, we propose a general architecture that captures the essential properties of superimposition that are common to all software languages. 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. Technically, FEATUREHOUSE relies 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

ICSE'09, May 16–24, 2009, Vancouver, Canada
978-1-4244-3452-7/09/$25.00 © 2009 IEEE

on attribute grammars.

We have used FEATUREHOUSE to demonstrate that our approach of software composition is indeed general. We have integrated several, very different languages into FEATUREHOUSE almost automatically, in particular Java, C#, C, Haskell, JavaCC, and XML. That is, FEATUREHOUSE can be used to compose software artifacts written in these languages. We did not need to extend the languages themselves, as would be necessary in related tools such as AHEAD [8] or FeatureC++ [4], which relieved us from a lot of tedious and error-prone implementation work.

The integration of a new language is almost entirely based on the language's grammar specification plus some attributes added as annotations and some rather small composition rules (usually not more than 10 to 20 LOC). We have applied FEATUREHOUSE to compose software systems of different sizes (1 KLOC–60 KLOC), written in different languages (Java, C#, C, Haskell, JavaCC, and XML). Our studies demonstrate the practicality and scalability of our approach and tools and provide insights 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 highlight the generality of superimposition as a composition technique.
2. We propose a general approach to software composition that is applicable to different languages.
3. We provide a framework and tool chain and report on experiments with six languages and eight software systems.
4. We initiate a discussion of language and granularity issues of software composition by superimposition.

2. FEATUREHOUSE

FEATUREHOUSE is a general architecture of software composition supported by a framework and tool chain. FEATUREHOUSE provides facilities for feature composition based on a language-independent model of software artifacts and an automatic plug-in mechanism for the integration of new artifact languages. FEATUREHOUSE generalizes and integrates a previous software composition tool for software composition, called FSTCOMPOSER [5], and improves over prior work on AHEAD in that it implements language-independent software composition; while AHEAD provides a language-independent *model* based on nested vectors, the support for different languages has been implemented for each language from scratch (see Sec. 6). The code of FEATUREHOUSE as well as examples and case studies can be downloaded from the project's website.[1]

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. Composition

FSTCOMPOSER relies on a general model of the structure of software artifacts, called the *feature structure tree (FST)* model. FSTs are designed to represent any kind of artifact with a hierarchical structure. An FST represents the essential modular structure of a software artifact and abstracts from language-specific details. For example, an artifact written in Java contains packages, classes, methods, etc., which are represented by nodes in the FST. An XML document (e.g., XHTML) may contain elements that represent the underlying document structure, e.g., headers, sections, paragraphs. A makefile or build script consists of definitions and rules that may be nested.

Each node of an FST has (1) a name that corresponds to the name of the artifact's structural element it represents and (2) a type that corresponds to the syntactical category the structural element belongs to. For example, a class Foo is represented by a node Foo of type class. Essentially, an FST is a stripped-down abstract syntax tree (AST): it contains only information that is necessary for the specification of the modular structure of an artifact. The inner nodes of an FST denote modules (e.g., classes and packages) and the leaves store the modules' content (e.g., method bodies and field initializers). We call the inner nodes *non-terminals* and the leaves *terminals*. For illustration, Figure 1 depicts an excerpt of a class of a database system taken from one of our case studies (Sec. 4). The complete class is located in a subpackage structure and contains 13 fields, 2 constructors, 58 methods, and 4 inner classes.

The answer to the question of which structural elements are represented as inner nodes and leaves, respectively, depends on the degree of *granularity* at which software artifacts are to be composed [23]. In our example, different granularities would be possible, e.g., we could represent only packages and classes but not methods or fields as FST nodes (a coarse granularity), or we could also represent statements or expressions (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. However, we have made the experience that the granularity of Figure 1 is usually sufficient for composition. We will return later to this issue.

The composition of software artifacts proceeds by the superimposition of the corresponding FSTs, 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. Figure 2 illustrates the process of FST superimposition with our database example. The artifact BASEDB is superimposed with an

[1] http://www.fosd.de/fh

```
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 }
```

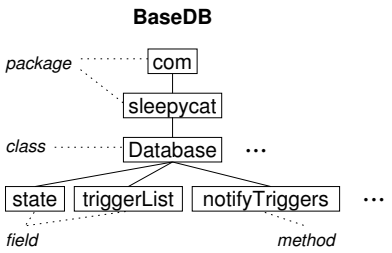


Figure 1. Java code and FST of the artifact BASEDB**.**

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 the method notifyTriggers of BASEDB via overriding (the keyword original defines how two method bodies are composed, which is similar to Java's super).

Generally, the composition of two leaves of an FST that contain further content, e.g., the two bodies of notifyTriggers demands a special treatment. The reason is that the content is not represented as a subtree but as plain text. For example, method bodies are composed differently from fields, XML text elements, or JavaCC grammar productions. The solution is that, depending on the artifact language and node type, different rules for composition are used. Often simple rules like replacement, concatenation, specialization, or overriding suffice, but the approach is open to more sophisticated rules known from multi-dimensional separation of concerns [34] or software merging [29]. For example, in our case studies, we merge two methods bodies via overriding, where original defines how the bodies are merged. Note that original is not a new keyword added to Java but only a meta-notation that disappears after composition. The Java parser treats original like a method call. During the composition of two method bodies, original is searched and substituted for the original method body modulo some renaming. This requires not to use original as name for ordinary methods.

Technically, multiple software artifacts (e.g., code and corresponding documentation) can be grouped in a so-called *unit of composition*. 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 non-terminals and the files located inside a subdirectory as terminals. Of course, if we intend to achieve a finer composition granularity (e.g., at the level of XML elements, classes and methods, and grammar rules), we can add further levels of non-terminals representing the artifacts' substructures, as we will explain next.

2.2. Generation and Automation

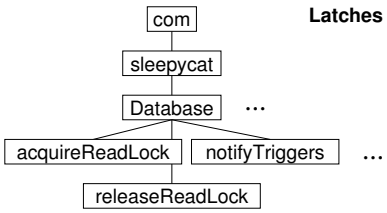
FEATUREHOUSE is a general framework and tool chain for software composition into which new languages can be plugged easily. The idea is that, although artifact languages are very different, the process of software composition by superimposition is very similar. Previous approaches that rely on superimposition did not take advantage of this similarity. For example, the developers of AHEAD [8] and FeatureC++ [4] have extended the artifact languages Java and C++ by constructs and mechanisms for composition. They have each implemented a parser, a superimposition algorithm, and a pretty printer[2] – all specific to the artifact language. In our previous work, we have introduced the FST model in order to be able to express superimposition independently of an artifact language [5]. Nevertheless, we had to provide for each language, in particular, for Java, C#, and XML:

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

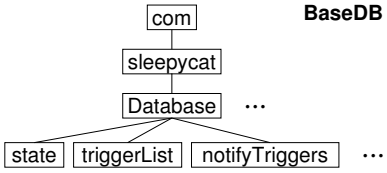
[2]With 'pretty printer' we refer to a tool that takes a parse tree or an FST and generates source code.

```
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 }
```

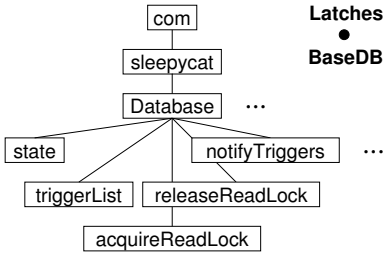


Figure 2. Java code of LATCHES • BASEDB**.**

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 that we spent lots of time on debugging.

2.2.1. FSTGENERATOR

These problems motivated us to pursue the automation of the integration of an additional language and base it largely on the language's grammar. 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 in Section 2.2.2. We have developed a tool, called FSTGENERATOR, that generates almost all code that is necessary for the integration of a new language. FSTGENERATOR expects the grammar of the language in a proprietary format, called FEATUREBNF. FEATUREBNF is similar to the Backus-Naur-Form but supports some extensions [40] and annotations, some of which are used by FSTGENERATOR, as we will explain. Using a grammar written in FEATUREBNF, FSTGENERATOR generates an LL(k) parser that directly produces FST nodes and 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; (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., a rule for method overriding or for the concatenation of the statements of two constructors. Figure 3 illustrates the interplay between FSTGENERATOR and FSTCOMPOSER and Table 1 lists some examples of composition rules.

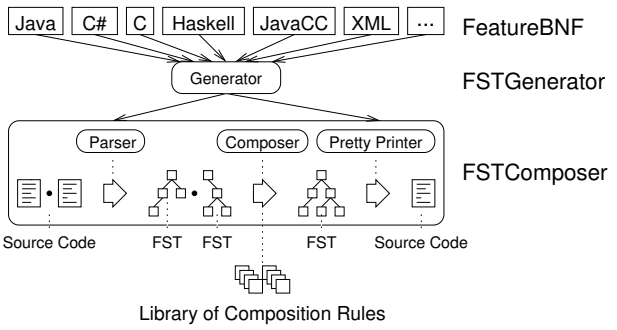


Figure 3. The architecture of FEATUREHOUSE.

rule	description
method overriding	merges two method bodies; original is used to inline one body into the other
grammar rule overriding	merges two grammar rules; original is used to inline the body of one rule into 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

Table 1. Examples of composition rules.

2.2.2. Attributes

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

As mentioned, without any attributes, FSTGENERATOR would create, for each production rule, a corresponding terminal node (e.g., for a class declaration rule a node is created that stores information of that declaration), and only the top-level terminals would appear in the generated FST; in our case, beside a non-terminal denoting the enclosing Java file, there would only be a terminal node per class declaration, and the class' member declarations would appear as text in the terminal's content. Since this granularity is too coarse for our purposes, we use attributes to annotate those production rules that are non-terminals, i.e., that contain further nodes.

Figure 5 depicts an annotated version of our simple Java grammar. The attribute @FSTNonTerminal above the rule ClassDecl states that classes are non-terminals that contain further elements. It follows from the grammar that a node representing a class may have children representing its name, supertype, fields, constructor, and methods. The attribute's parameter name is used to assign the name of a class to the FST node representing the class.

```
1 ClassDecl : "class" Type "extends" ExtType "{"
2   (VarDecl)* (ClassConstr)* (MethodDecl)*
3 "}";
4 VarDecl : Type <IDENTIFIER> ";";
5 MethodDeclaration :
6   Type <IDENTIFIER> "(" (FormalParamList)? ")" "{"
7     "return" Expression ";"
8   "}";
```

Figure 4. An excerpt of a simplified Java grammar.

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

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

With a single attribute, we have refined the composition granularity of Java artifacts. Now, Java FSTs have three levels (omitting packages and imports): (1) a root that represents the Java file, (2) classes that are non-terminals, and (3) type names, methods, constructors, and fields that are terminals. Without the attribute of Figure 5, Java FSTs have only two levels: (1) a root and (2) classes. In Figure 6, we show the difference between the two levels of granularity.

Beside @FSTNonTerminal, FSTGENERATOR supports several further attributes. For example, the attribute @FSTTerminal is used to mark terminal nodes. Although all production rules that are not annotated are interpreted as terminals, this attribute allows a programmer to define the format of the name that appears in the FST node and

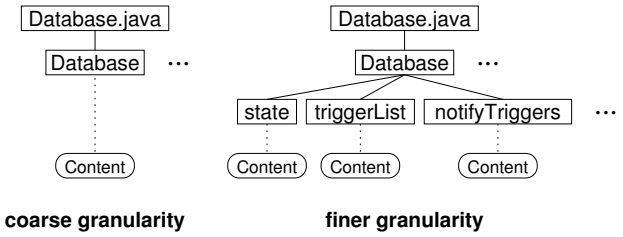


Figure 6. Two granularities of representing Java artifacts as FSTs.

the composition rule for merging the content of two corresponding terminals. For example, in Figure 7, we specify that the name of a node representing a Java method receives the method's name (<IDENTIFIER>) followed by its formal parameters (FormalParamList). Production rules without explicitly assigned names receive proper standard names.

Note that, without the possibility to specify the name of an FST node, in many cases a superimposition would not be feasible. Recall that two nodes are superimposed only if their names (and types) are identical. For example, we can use the 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}"). Due to the lack of space, we omit a description of the pattern language for the specification of FST node names.

Using the parameter compose, we define which composition rule from the library is used when composing terminal nodes. In our simple Java grammar, we define that methods are composed via method overriding, hence, 'JavaMethodOverriding' refers to an artifact-specific composition rule that is part of the library of composition rules (see Fig. 3).

```
1 @FSTTerminal(name="{<IDENTIFIER>}({FormalParamList})",
2   compose="JavaMethodOverriding")
3 MethodDecl :
4   Type <IDENTIFIER> "(" ( FormalParamList )? ")" "{"
5     "return" Expression ";"
6   "}";
```

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

3. Integrating Languages

Next we report on some observations we made while integrating languages into FEATUREHOUSE. First, we discuss our general observations and, subsequently, we compare the FEATUREHOUSE approach, which we call *generative approach*, with the approach taken previously by FSTCOMPOSER, which we call *manual approach*. In Table 2, we provide a summary of the languages that we have integrated, listing the overall number of grammar rules, the number of rules annotated to represent non-terminals, the number of rules annotated to represent terminals, and the overall number of attributes.

3.1. General Observations

Language properties. From our experience with the integration of artifact languages and from theoretical discussions [6], we have inferred four properties that a language must have in order to be ready for superimposition and 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 syntactical category that becomes the node's type.
3. An element must not contain two or more direct child elements with the same name and type.
4. Elements that do not have a hierarchical substructure represented in the FST (terminals) must provide composition rules, or must not be composed.

The first property implies that it would even be possible to compose unstructured text when it is considered as a single terminal node. In this case, two text nodes would be merged via string concatenation. The more structure is exposed in an FST, the more fine-grained the composition can be (which typically makes the composition more expressive and easier to implement), but there is usually a natural limit. For example, representing and superimposing arithmetic expressions is certainly not useful because expressions do not have unique identifiers.

	Java	C#	C	Haskell	JavaCC	XML
# rules	135	229	45	78	170	14
# non-terminals	10	17	2	13	16	6
# terminals	13	18	9	9	16	6
# attributes	42	53	21	24	61	15

Table 2. Overview of the languages integrated in FEATUREHOUSE.

Exceptions. We have found that the languages and granularities we looked at have the properties shown above, with one exception that demanded a special treatment: XML is a template for languages. Simply parsing and composing XML elements on the basis of the tags' names is oftentimes not appropriate. For example, in XHTML, we cannot compose the elements <li> of a list <ul> by name. The reason is that, in general, the elements of <ul> do not have unique names, which contradicts the third of our properties. To solve this problem, we have added the possibility to assign a specific name attribute to each XML tag. A more elegant solution would be to annotate the XML grammar, so that, for instance, for each list entry, a unique name is generated. Actually, an XML schema defines the meaning of the different elements of an XML language. Consequently, we would have to annotate the XML schema directly. In this sense, an annotated XML schema would play the same role as an annotated FEATUREBNF grammar.

Effort. For every language, we were able either to transform an existing grammar in JavaCC or ANTLR to FEA-

TUREBNF (as we did for Java, C#, and JavaCC) or to write our own grammar (as we did for C, Haskell, and XML). In the former cases, we spent usually only few hours. The main issue in the transformation was the conversion of nested ANTLR or JavaCC production rules into flat FEATUREBNF production rules. This is a technical limitation in generating a pretty printer without overhead. Writing grammars from scratch was more time-consuming – usually around one or two days – but was reduced by parsing only the portion of an artifact's content that is relevant for the FST.

A further, interesting observation was that annotating the C# grammar was more complicated and time-consuming than annotating the Java grammar. There are two reasons for this: (1) C# contains more language constructs, and (2) the developers of the original ANTLR grammar we used for C# had the goal to minimize the lookahead, which results in a more complex specification.

Generality. Although they are not object-oriented, we were able to integrate C and Haskell well into FSTCOMPOSER. However, neither of the two languages offers many candidates for non-terminals. C is an imperative language that is built around structures and procedures, and Haskell is a functional language that is based on data types and functions. We represented C and Haskell files, modules, and data structures (struct, data) as non-terminals and their declarations such as functions and typedefs as terminals.

We were even able to integrate the language JavaCC into FSTCOMPOSER. As a basis, we used a grammar, written itself in JavaCC, and translated it to FEATUREBNF. A peculiarity of JavaCC is that it contains rules for grammar specification and rules for embedded Java code. For the Java part, we reused our existing solution. Annotating the grammar part was straightforward: the overall grammar specification is a non-terminal that consists of several terminals representing tokens and production rules.

3.2. Manual vs. Generative Approach

Granularity. 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 non-terminals and terminals. In the generative approach, the attributes of the grammar define which structural elements are represented by non-terminals and terminals. Changing the attributes is a matter of minutes; changing the adapter is tedious and error-prone. Hence, the generative approach enabled us to experiment with different granularities in the first place. For example, in Haskell, it was not clear which degree of granularity of superimposition is appropriate. It was clear that function definitions are terminals but not whether data type definitions are terminals or non-terminals. After playing with some examples, we realized that it is quite useful to represent data types as non-terminals, so that data type definitions can be extended by adding new type constructors (see Sec. 4), e.g.:

	manual approach			generative approach	
	adapter	pretty printer	comp. rules	comp. rules	attributes
Java	1366	424	214	178	42
C#	2851	374	518	53*	53
XML	454	75	42	14	15

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

Table 3. Amount of boilerplate code (LOC).

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

Boilerplate code. In the generative approach, we minimize the amount of boilerplate code a programmer has to write. In Table 3, we list the amount of code we had to write in the manual and the generative approach for integrating Java, C#, and XML. We did not count generated code and the code of the language specifications, which were often publicly available. On average, the generative approach reduces the need for code writing to 6 % of that in the manual approach.

Composition rules. Based on our experience, we have extracted some terminal composition rules that we collected in a library. These rules were reused in the integration of several languages. Specifically, we have implemented composition rules for constructor concatenation, method overriding, field overriding, grammar rule overriding, implements list merging, modifier specialization, replacement, and text content concatenation (see Tab. 1). In some cases, we were able to reuse the implementation of a rule for merging the content of different types of terminals, e.g., the method overriding rule is used for merging Java methods and C functions. Via attributes we can specify declaratively for which kind of terminal we use which composition rule from the library, e.g., compose="JavaMethodOverriding" (see Sec. 2.2).

Expenditure of time. As mentioned in Section 3.1, in the generative approach, the effort of the integration of a new language was on the order of hours. In the manual approach, the effort was higher – on the order of days.

Susceptibility to error. Finally, writing a parser, pretty printer, and adapter code is tedious and 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 neglected to represent inner classes as non-terminals. In the generative approach, we stumbled early over this issue since it was exposed by the grammar because we annotated grammars top-down, starting from the root production and, at some point, we reached inner classes and interfaces. Another example are (static) initializers of Java classes. In the manual integration, we simply did not think of the possibility to merge them, until needed in a case study (Berkeley DB; see Sec. 4). When annotating the grammar top-down, this option became obvious.

4. Composing Software Systems

In order to demonstrate the practicality of FEATUREHOUSE, we have composed software systems of different sizes written in different languages. In Table 4, we summarize information on the software systems and their compositions. We highlight here only some interesting observations. The source code of all software systems of our study can be downloaded at the FEATUREHOUSE website.

Scalability. In order to learn about scalability, we used FEATUREHOUSE to compose a variant of Berkeley DB, which we refactored before into superimposeable units using a product line tool [23]. Overall, we have composed 99 units of composition of Berkeley DB with almost 60 thousand lines of code. Our study shows that, even though FEATUREHOUSE is an unoptimized prototype, it scales well to medium-sized software projects (24 sec. composition time).

Generality. Despite some subtle differences, the Java and C# versions of the Graph Product Line (GPL) are very similar before *and* after composition. This demonstrates the generality of superimposition and the necessity for a language-independent model and tool chain.

Furthermore, we confirmed that superimposition is indeed not limited to object-oriented languages. Decomposing the C program GraphLib demonstrated that, even at the level of pure functions, superimposition is an appropriate composition mechanism. However, we noted also a problem: sometimes we wanted to add a new #include statement in front of a file or between other #include statements. This is a problem since the order of #include statements matters. Using superimposition, we had to divide some units of composition into a part that adds the #include statement and a part that extends the code below in the file. By subdividing features, which is a kind of Parnas' sandwiching [35], we were able to control the order of #include statements in the composed C code. This problem does not occur with Java's import statements or C#'s using statements because the Java and C# grammars prescribe that these statements have to appear before the class declarations and the statements' order does not matter. In this sense, at the chosen granularity, Java and C# are better suited for superimposition than C is. For C, the problem was manageable since it occurred only a few times.

We were also able to support and compose Haskell artifacts. As with C, it was straightforward to merge files containing different data type declarations and functions. However, in Haskell, a function definition may have a signature and multiple equations that are distinguished by their argument lists. In contrast to Java or C#, the order in which the equations appear in a Haskell file affects the execution order, since some patterns of arguments may overlap. Basically, we have the same problem as with C's #include, and we solved the problem in the same way. Nevertheless, such a language feature does not abet superimposition.

Non-code artifacts. We found some use cases for composing non-code artifacts. We have extended 9 units of composition of GPL with a simple documentation in form of a basic XHTML file and several refinements that extend the documentation. We have used FEATUREHOUSE successfully to generate documentations based on selections of composition units. As mentioned in Section 2.1, the superimposition of XHTML documents required special preparation: we had to assign to each tag (e.g., a list or a section) that we wanted to extend (to merge with another tag) a unique identifier that has been used during superimposition.

Furthermore, on the language Feature Featherweight Java (FFJ) [3], we have demonstrated the capability to compose JavaCC grammars. In a different line of work, we discussed type systems for different extensions of Featherweight Java (FJ) [3]. In order to let the FJ parser recognize the new syntactical elements of FFJ, we added and overrode several JavaCC production rules. For example, we added new keywords for class refinements and method refinements.

Finally, Violet has 83 units of composition that contain, in summary, 98 property files. A property file stores text-based configuration information of the UML editor, e.g., edge1.tooltip=Association. Individual composition units of Violet provide individual configuration information. We have successfully used FEATUREHOUSE to compose property files, thus, even artifacts with unstructured content align well with our approach.

	COM	CLA	LOC	TYP	TIM	Description
FFJ	2	–	289	JavaCC	< 1 s	Feature Featherweight Java Grammar [3]
Arith	15	–	442	Haskell	< 1 s	Arithmetic expression evaluator
GraphLib	13	–	934	C	1 s	Low-level graph library[a]
GPL	26	57	2,439	Java, XML	2 s	Graph Product Line (Java Version) [26]
GPL	26	57	2,148	C#	2 s	Graph Product Line (C# Version)
Violet	88	157	9,660	Java, Text	7 s	UML Editor[b]
GUIDSL	26	294	13,457	Java	10 s	Configuration management tool [7]
Berkeley DB	99	765	58,030	Java	24 s	Oracle's Embedded Storage Engine [23]

COM: number of units of composition; CLA: number of classes; LOC: lines of code; TYP: types of contained artifacts; TIM: time to compose

[a] http://keithbriggs.info/graphlib.html [b] http://www.horstmann.com/violet/

Table 4. Overview of the case studies.

5. Lessons Learned

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

1. Superimposition of FSTs scales to medium-sized software projects.
2. The time for annotating grammars is moderate and the depths of the generated FSTs depend on the composition granularity and the complexity of the language.
3. Artifacts written in languages whose structural elements may have identical names (whose elements are distinguished by the lexical order) have to prepared, in order to be superimposed. Basically, each FST node receives a unique name, as in GPL's XHTML documentation.
4. The order of an artifact's elements represented by FST nodes (terminals or non-terminals) may matter. If it matters, as in the case of #include, the technique of sandwiching can be used as a workaround.
5. Non-code or even unstructured artifacts, such as plain text files or binaries, can be integrated seamlessly with FSTCOMPOSER.
6. In practice, only a few composition rules are needed, which can be reused by different languages and which follow even fewer rule patterns.
7. Superimposition is applicable to a wide range of code and non-code languages including object-oriented languages, functional languages, imperative languages, document description languages, and grammar specification languages.

6. Related Work

Although manifested and implemented differently, several languages provide support for superimposition of different kinds of artifacts, e.g., *Jiazzi* [28], *Classbox/J* [9], Hyper/J [34] and *Jak* [8] for Java, *ContextL* [15] for Lisp, *FeatureC++* [4] for C++, Xak [1] for XML, and others [8, 10, 14, 16, 21, 37].

While it has been noted that there is a unique core of all composition mechanisms based on superimposition [8], 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 [8]. 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, in the FST model, we distinguish between terminals and non-terminals and presume a fixed order of elements. This and the tree structure poses the FST model closer to the implementation level and allowed us to derive directly an implementation, which was not done in the case of AHEAD (each AHEAD tool for each language has been developed from scratch). So, we believe that our FST model captures more precisely the essence of superimposition. 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 [6]) 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 language-independently.

We have developed an algebra and two calculi of feature composition which are consistent with the FST model [2, 3, 6]. They allow us to explore general properties of software composition and typing issues. Results of these projects have been incorporated into FEATUREHOUSE, e.g., the fact that superimposition is associative. Similarly, practical issues arising from composing software systems with FEATUREHOUSE had an impact on theory, e.g., the observation that the degree of composition granularity influences the algebraic properties of composition.

Beside superimposition, also other composition techniques have been proposed. For example, composition by quantification, as used in aspect-oriented programming [27], is a frequently discussed technique. In the context of our FST model, quantification can be modeled as a tree walk [6], in which each node is visited and a predicate specifies whether the node is modified or not. Harrison et al. [19] propose a sophisticated set of rewriting rules that are based on tree walks.

Recent work in model composition [11, 17, 25] aims at developing a general framework for composing different kinds of models. Our approach is even more general in that it aims also at non-modeling languages. In preliminary work, we have used FEATUREHOUSE to compose UML class and state diagrams serialized in XMI files.

Software composition is related to the broad field of software merging. Software merging attempts 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 [29]. 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 a legacy software system into a product line, to type-check *all* products of a product line, and to visualize and resolve feature interactions [22, 23]. CIDE pursues also a generative approach of integrating new languages [24] based on the same grammar format we use in FEATUREHOUSE but on different attributes; initially, FEATUREBNF has been developed for CIDE. It 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 refactor Berkeley DB, one of our case studies.

7. Conclusion

We model software artifacts by tree structures and composition by tree superimposition. 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.

We have presented a general approach to software composition based on FST superimposition, and we offer a set of accompanying tools integrated in the FEATUREHOUSE framework. 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 superimposition. From the integration of various 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 (cf. Sec. 5).

Presently, we are working on a formalization of the FST model and on further tools to be integrated into FEATUREHOUSE that operate on FSTs, e.g., a tool that visualizes FSTs and a tool that analyzes interactions between pieces of software. Furthermore, we are working on support for type checking and safe composition, and we intend to integrate other composition techniques such as quantification and aspect weaving. We have reason to believe that these align well with the FST model [6]. Finally, we will explore how XML languages can be integrated better. In this respect, we will evaluate whether an XML schema plus attributes can play the role of FEATUREBNF.

Acknowledgments

We thank 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, Abhinay Kampasi for refactoring Violet into composeable units, and Marko Rosenmüller and Norbert Sigmund for their support in developing the C grammar. This work has been supported in part by the German Research Foundation (DFG), project number AP 206/2-1.

References

[1] F. Anfurrutia, O. Díaz, and S. Trujillo. On Refining XML Artifacts. In *Proc. Int'l. Conf. Web Engineering*, volume 4607 of *LNCS*, pages 473–478. Springer-Verlag, 2007.

[2] S. Apel and D. Hutchins. An Overview of the gDeep Calculus. Technical Report MIP-0712, University of Passau, 2007.

[3] S. Apel, C. Kästner, and C. Lengauer. Feature Featherweight Java: A Calculus for Feature-Oriented Programming and Stepwise Refinement. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, pages 101–112. ACM Press, 2008.

[4] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 125–140. Springer-Verlag, 2005.

[5] S. Apel and C. Lengauer. Superimposition: A Language-Independent Approach to Software Composition. In *Proc. Int'l. Symp. Software Composition*, volume 4954 of *LNCS*, pages 20–35. Springer-Verlag, 2008.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Proc. Int'l. Conf. Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 36–50. Springer-Verlag, 2008.

[7] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l. Software Product Line Conf.*, volume 3714 of *LNCS*, pages 7–20. Springer-Verlag, 2005.

[8] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Engineering*, 30(6):355–371, 2004.

[9] A. Bergel, S. Ducasse, and O. Nierstrasz. Classbox/J: Controlling the Scope of Change in Java. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 177–189. ACM Press, 2005.

[10] L. Bergmans and M. Aksit. Composing Crosscutting Concerns Using Composition Filters. *Comm. ACM*, 44(10):51–57, 2001.

[11] P. Bernstein, A. Halevy, and R. Pottinger. A Vision for Management of Complex Models. *SIGMOD Rec.*, 29(4):55–63, 2000.

[12] J. Bosch. Super-Imposition: A Component Adaptation Technique. *Information and Software Technology*, 41(5):257–273, 1999.

[13] L. Bouge and N. Francez. A Compositional Approach to Superimposition. In *Proc. Int'l. Symp. Principles of Programming Languages*, pages 240–249. ACM Press, 1988.

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

[15] P. Costanza, R. Hirschfeld, and W. de Meuter. Efficient Layer Activation for Switching Context-Dependent Behavior. In *Proc. Joint Modular Languages Conf.*, volume 4228 of *LNCS*, pages 84–103. Springer-Verlag, 2006.

[16] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, volume 3676 of *LNCS*, pages 422–437. Springer-Verlag, 2005.

[17] J. Dingel, Z. Diskin, and A. Zito. Understanding and Improving UML Package Merge. *Software and Systems Modeling*, 7(4):443–467, 2008.

[18] G. Freeman, D. Batory, and G. Lavender. Lifting Transformational Models of Product Lines: A Case Study. In *Proc. Int'l. Conf. Model Transformation*, volume 5063 of *LNCS*, pages 16–30. Springer-Verlag, 2008.

[19] W. Harrison, H. Ossher, and P. Tarr. General Composition of Software Artifacts. In *Proc. Int'l. Symp. Software Composition*, volume 4089 of *LNCS*, pages 194–210. Springer-Verlag, 2006.

[20] D. Hutchins. Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 1–19. ACM Press, 2006.

[21] T. Kamina and T. Tamai. Lightweight Scalable Components. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, pages 145–154. ACM Press, 2007.

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

[23] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l. Conf. Software Engineering*, pages 311–320. ACM Press, 2008.

[24] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Language-Independent Safe Decomposition of Legacy Applications into Features. Technical Report 02/2008, University of Magdeburg, 2008.

[25] D. Kolovos, R. Paige, and F. Polack. Merging Models with the Epsilon Merging Language (EML). In *Proc. Int'l. Conf. Model Driven Engineering Languages and Systems*, volume 4199 of *LNCS*, pages 215–229. Springer-Verlag, 2006.

[26] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. Int'l. Conf. Generative and Component-Based Software Engineering*, volume 2186 of *LNCS*, pages 10–24. Springer-Verlag, 2001.

[27] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 2743 of *LNCS*, pages 2–28. Springer-Verlag, 2003.

[28] S. McDirmid and W. Hsieh. Aspect-Oriented Programming with Jiazzi. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 70–79. ACM Press, 2003.

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

[30] M. Mezini and K. Ostermann. Conquering Aspects with Caesar. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 90–100. ACM Press, 2003.

[31] N. Nystrom, S. Chong, and A. Myers. Scalable Extensibility via Nested Inheritance. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 99–115. ACM Press, 2004.

[32] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 41–57. ACM Press, 2005.

[33] H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In *Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications*, pages 25–40. ACM Press, 1992.

[34] H. Ossher and P. Tarr. Hyper/J: Multi-Dimensional Separation of Concerns for Java. In *Proc. Int'l. Conf. Software Engineering*, pages 734–737. IEEE CS Press, 2000.

[35] D. Parnas. Designing Software for Ease of Extension and Contraction. In *Proc. Int'l. Conf. Software Engineering*, pages 264–277. IEEE CS Press, 1978.

[36] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 419–443. Springer-Verlag, 1997.

[37] M. Sihman and S. Katz. Superimpositions and Aspect-Oriented Programming. *Computer J.*, 46(5):529–541, 2003.

[38] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Software Engineering and Methodology*, 11(2):215–255, 2002.

[39] P. Tarr, H. Ossher, W. Harrison, and S. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proc. Int'l. Conf. Software Engineering*, pages 107–119. IEEE CS Press, 1999.

[40] D. Wile. Abstract Syntax from Concrete Syntax. In *Proc. Int'l. Conf. Software Engineering*, pages 472–480. ACM Press, 1997.