Superimposition: A Language-Independent Approach to Software Composition

Sven Apel and Christian Lengauer

Department of Informatics and Mathematics
University of Passau, Germany
{apel, lengauer}@uni-passau.de

Abstract. Superimposition is a composition technique that has been applied successfully in several areas of software development. In order to unify several languages and tools that rely on superimposition, we present an underlying language-independent model that is based on *feature structure trees (FSTs)*. Furthermore, we offer a tool, called FST-Composer, that composes software components represented by FSTs. Currently, the tool supports the composition of components written in Java, Jak, XML, and plain text. Three nontrivial case studies demonstrate the practicality of our approach.

1 Introduction

Software composition is the process of constructing software systems from a set of components. It aims at improving the reusability, customizability, and maintainability of large software systems.

One popular approach to software composition is superimposition. *Superimposition* is the process of composing software artifacts of different components by merging their corresponding substructures. For example, when composing two components, two internal classes with the same name, say `Foo`, are merged, and the result is called again `Foo`.

Superimposition has been applied successfully to the composition of class hierarchies in multi-team software development [1], the extension of distributed programs [2,3], the implementation of collaboration-based designs [4–6], feature-oriented programming [7,8], subject-oriented programming [9,10], aspect-oriented programming [11, 12], and software component adaptation [13]. All these approaches superimpose hierarchically organized program constructs by matching their levels, names, and types in the hierarchy.

It has been noted that, when composing software, not only code artifacts have to be considered but also noncode artifacts, e.g., documentation, grammar files, makefiles [8,10]. Thus, superimposition, as a composition technique, should be applicable to a wide range of software artifacts. While there are tools that implement superimposition for noncode artifacts [8, 14–19], they are specific to their underlying languages.

It is an irony that, while superimposition is such a general approach, up to now, it has been implemented for every distinct kind of software artifact from scratch. In our recent work, we have explored the essential properties of superimposition and developed an algebraic foundation for software composition based on superimposition [20].

We present a model of superimposition based on *feature structure trees (FSTs)*. An FST represents the abstract hierarchical structure of a software component. That is, it hides the language-specific details of a component's implementation. The nodes of an FST represent the structural elements of a component. However, an FST contains only nodes that represent the modular component structure (modules and submodules) and that are relevant for composition.

Furthermore, we have a tool, called FSTComposer, that implements composition by superimposition on the basis of FSTs. At present, FSTComposer is able to compose software components written in Java, Jak,[1] XML, and plain text. Three nontrivial case studies demonstrate the practicality and scalability of our approach and tool.

2 A Tree Representation of Software Artifacts

A software component is represented as an FST. The nodes of an FST represent a component's structural elements. Each node has a name,[2] which is also the name of the structural element that is represented by the node.

FSTs are designed to represent any kind of component with a hierarchical structure. For example, a component written in Java contains packages, classes, methods, etc., which are represented by nodes in the FST. An XML document (e.g., XHTML) may contain tags that represent the underlying document structure, e.g., chapters, sections, paragraphs. A makefile or build script consists of definitions and rules that may be nested.

An FST is a stripped-down abstract syntax tree: it contains only the information that is necessary for the specification of the structure of a component. The nature of this information depends on the degree of granularity at which software artifacts are to be composed [22], as we discuss below.

Principally, a component may contain elements written in different code and noncode languages, e.g., makefiles, design documents, performance profiles, mathematical models, diagrams, documentation, or deployment descriptors, which all can be represented as FSTs [8, 10]. While our work is not limited to code artifacts, for simplicity, we explain our ideas by means of Java.

Furthermore, type information is attached to the nodes. This is important during component composition in order to prevent the composition of incompatible nodes, e.g., the composition a field with a method.

The FSTs we consider are unordered trees. That is, the children of a node in an FST do not have a fixed order, much like the order of field declarations

[1] Jak is a Java-like language for stepwise refinement and feature-oriented programming [21]. It extends Java by the keyword `refines` in order to express subsequent class extensions.

[2] Mapped to specific component languages, a name could be a string, an identifier, a signature, etc.

in a Java class is irrelevant. However, some languages may require a fixed order (e.g., the order of sections in a text document matters). This will be addressed in further work.

Figure 1 depicts an excerpt of the implementation of a Java component BasicStack and its representation in form of an FST. The FST contains nodes that represent packages, classes, interfaces, fields, and methods, etc. They do not reflect information about the internal structure of methods or the variable initializers of fields. That is, our FST only represents the *modular substructure* of a software artifact (and not more). The structure and content of modules is not always modelled completely, e.g., our FST in Figure 1 does not represent the full Java abstract syntax tree including statements, parameters, or expressions, but only the main structural elements. A different granularity would be possible [22], e.g., we could represent only packages and classes but not methods or fields as FST nodes, or we could also represent statements or expressions. However, we will demonstrate that the granularity we chose is sufficient for composition, while it simplifies the overall process. At the same time, reasoning at a finer grain is still possible, i.e., method bodies can be composed without representing their substructure, as we will show in Section 3.2.

```
package util;
class Stack {
  LinkedList data = new LinkedList();
  void push(Object obj) {
    data.addFirst(obj);
  }
  Object pop() {
    return data.removeFirst();
  }
}
```

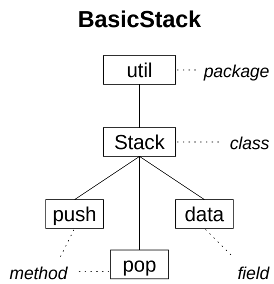

Fig. 1. Java code and FST of the component BasicStack.

3 Component Composition by FST Superimposition

Superimposition is the process of composing trees recursively by composing nodes at the same level (counting from the root) with the same name[3] and type. Our aim is to abstract from the specifics of present tools and languages and to make superimposition available to a broader range of software artifacts. Moreover, a general model allows us to study the essence of software composition by superimposition, apart from language- and tool-specific issues. Our work is motivated by the observation that, principally, composition by superimposition

[3] Of course, the use of aliasing techniques would allow a programmer to compose artifacts that have different names [23].

is applicable to any kind of software artifact that provides a sufficient structure [8, 10], i.e., a structure that can be represented as an FST.

With superimposition, two trees are composed by composing their corresponding nodes, starting from the root and proceeding recursively. Two nodes are composed to form a result node (1) when their parents (if there are parents) have been composed, i.e., they are on the same level, and (2) when they have the same name and type. The result node receives the name and type of the nodes that have been composed. Some nodes (the leaves of an FST) have also content, which is composed as well (see Sec. 3.2). If two nodes have been composed, the process of composition proceeds with their children. If a node has no counterpart to be composed with, it is added as separate child node to the composed parent node. This recurses until all leaves have been reached.

In Figure 2, we list a Java function `compose` that implements recursive composition. In Line 2, two nodes are composed, which succeeds only when the nodes are compatible (same name and type). In the case that the two nodes are terminals, their content is composed as well. In Lines 4–9, all children of the input trees (which are in fact subtrees) are composed recursively. That is, for each node in `treeA`, `findChild` returns the corresponding node in `treeB`, if there is one. Then, in Lines 8 and 10–13, the remaining nodes that have no counterpart to be composed with are added to the new parent node.

```
1  static Tree compose(Tree treeA, Tree treeB) {
2    Node newNode = treeA.node().composeNode(treeB.node());
3    if(newNode != null) {
4      Tree newTree = new Tree(newNode);
5      for(Tree childA : treeA.children()) {
6        Tree childB = treeB.findChild(childA.name(),childA.type());
7        if(childB != null) newTree.addChild(compose(childA, childB));
8        else newTree.addChild(childA.copy());
9      }
10     for(Tree childB : treeB.children()) {
11       Tree childA = treeA.findChild(childB.name(),childB.type());
12       if(childA == null) newTree.addChild(childB.copy());
13     }
14     return newTree;
15   } else return null;
16 }
```

Fig. 2. A Java function for composing FSTs.

Figure 3 illustrates the process of FST superimposition with a Java example; Figure 4 depicts the corresponding Java code. Our component BasicStack is composed with a component TopOfStack. The result is a new component, which is called CompStack$_1$, that is represented by the superimposition of the FSTs of BasicStack and TopOfStack. The nodes `util` and `Stack` are composed with their counterparts, and their subtrees (i.e., their methods and fields) are composed in turn (i.e., are merged).

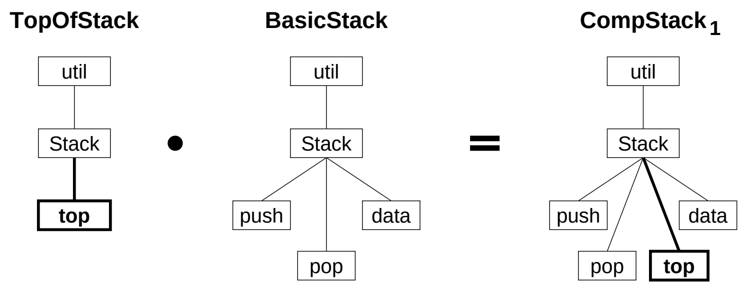

Fig. 3. FST superimposition of TopOfStack • BasicStack = CompStack$_1$.

3.1 Terminal and Nonterminal Nodes

Independently of any particular language, an FST is made up of two different kinds of nodes:

Nonterminal nodes are the inner nodes of an FST. The subtree rooted at a nonterminal node reflects the structure of some implementation artifact of a component. The artifact structure is *transparent* and subject to the recursive composition process. That is, a nonterminal node has only a name and a type, and no further content.

Terminal nodes are the leaves of an FST. Conceptually, a terminal node may also be the root of some structure, but this structure is *opaque* in our model. The substructure of a terminal does not appear in the FST. That is, a terminal node has a name, a type, and content.

While the composition of two nonterminals continues the recursive descent in the FSTs to be composed, the composition of two terminals terminates the recursion and requires a special treatment. There is a choice of whether and how to compose terminals:

Option 1: Two terminal nodes with the same name and type *cannot* be composed, i.e., their composition is considered an error.

Option 2: Two terminal nodes with the same name and type *can* be composed in some circumstances; each type has to provide its own rule for composition (see Sec. 3.2).[4]

In Java FSTs, packages, classes, and interfaces are represented by nonterminals. The implementation artifacts they contain are represented by child nodes, e.g., a package contains several classes and classes contain inner classes, methods, and fields. Two compatible nonterminals are composed by composing their child nodes, e.g., two packages with equal names are merged into one package that contains the composition of the child elements (classes, interfaces, subpackages) of the two original packages.

[4] Note that it would also be possible to provide specific rules for nonterminal composition, but we did not encounter this case so far.

```java
package util;
class Stack {
  Object top() { return data.getFirst(); }
}
```

$$\bullet$$

```java
package util;
class Stack {
  LinkedList data = new LinkedList();
  void push(Object obj) { data.addFirst(obj); }
  Object pop() { return data.removeFirst(); }
}
```

$$=$$

```java
package util;
class Stack {
  LinkedList data = new LinkedList();
  void push(Object obj) { data.addFirst(obj); }
  Object pop() { return data.removeFirst(); }
  Object top() { return data.getFirst(); }
}
```

Fig. 4. Java code of TopOfStack $\bullet$ BasicStack $=$ CompStack$_1$.

Java methods, fields, imports, modifier lists, and `extends`, `implements`, and `throws` clauses are represented by terminals (the leaves of an FST), at which the recursion terminates. Their inner structure or content is not considered in the FST model, e.g., the fact that a method contains a sequence of statements or that a field refers to a value or an expression.

Note that the first option of disallowing terminal composition [1] prevents method extension. But method extension is common practice in many approaches of software composition [6, 8, 10, 24–28]. Therefore, we choose the second option: providing language-specific composition rules for composing terminal nodes.

3.2 Composition of Terminals

In order to compose terminals, each terminal type has to provide its own rule for composition. Here are seven examples for Java-like languages:

- Two methods are composed if it is specified how the method bodies are composed (e.g., by overriding and using the keywords `original` [27] or `Super` [8] inside a method body).
- Two fields are composed by replacing one value with the value of the other or by requiring that one has a value assigned and the other has not.
- Two `implements` clauses are composed by concatenating their entries and removing duplicates.
- Two `extends` clauses are composed by replacing one entry with another entry (in the case of single inheritance) or by concatenating their entries and removing duplicates (in the case of multiple inheritance).

– Two `throws` clauses are composed by concatenating their entries and removing duplicates.
– Two modifier lists are composed by replacement following certain rules, e.g., `public` may replace `private`, but not vice versa.
– Two import declaration lists are composed by concatenating their entries and removing duplicates.

Overall, in Java-like languages, there are three kinds of composition rule patterns: overriding (methods), replacement (fields, `extends` clauses, modifier lists), and concatenation (imports, `implements` and `throws` clauses).

Figures 5 and 6 depict how Java methods are composed during the composition of the two features EmptyCheck and BasicStack using a *wrapping* composition rule. The methods `push` of EmptyCheck and BasicStack are composed in CompStack$_2$ by one method (`push`) wrapping the other (`push_wrappee`). The two `pop` methods are composed analogously. The keyword `original` [27],[5] provides a means to specify (without knowledge of their source code) how method bodies are merged. This composition rule is also applicable to other types and languages [8,14]. Other composition rules for composing method bodies, such as inlining would be possible.

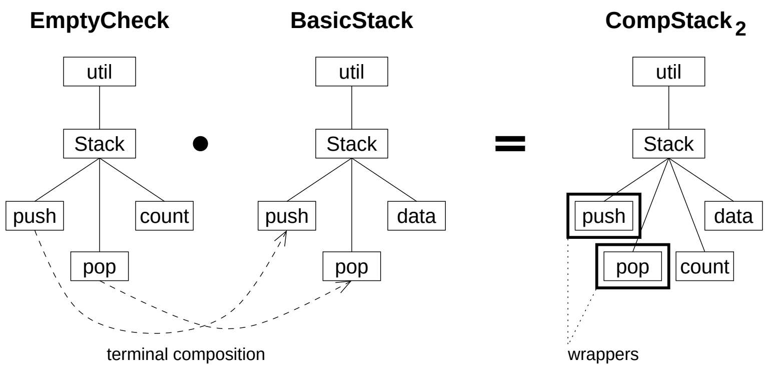

Fig. 5. Composing Java methods (FST representation).

Harrison et al. [23] propose a catalog of more sophisticated composition rules that permit a quantification over and a renaming of the structural elements of components. We argue that their rules are not specific to Java and can be reused to compose components written in other languages.

3.3 Discussion

Superimposition of FSTs requires several properties of the language in which the elements of a component are expressed:

[5] In the composed variant, `original` is replaced by a call to the wrapper.

```java
package util;
class Stack {
  int count = 0;
  void push(Object obj) { original(obj); count++; }
  Object pop() {
    if(count > 0) { count--; return original(); } else return null;
  }
}
```

$$\bullet$$

```java
package util;
class Stack {
  LinkedList data = new LinkedList();
  void push(Object obj) { data.addFirst(obj); }
  Object pop() { return data.removeFirst(); }
}
```

$$=$$

```java
package util;
class Stack {
  int count  = 0;
  LinkedList data = new LinkedList();
  void push_wrappee(Object obj) { data.addFirst(obj); }
  void push(Object obj) { push_wrappee(obj); count++; }
  Object pop_wrappee() { return data.removeFirst(); }
  Object pop() {
    if(count > 0) { count--; return pop_wrappee(); } else return null;
  }
}
```

Fig. 6. Composing Java methods.

1. The substructure of a component must be hierarchical, i.e., an n-ary tree.
2. Every element of a component must provide a name that becomes the name of the node in the FST.
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 (terminals) must provide composition rules, or cannot be composed.

These constraints are usually satisfied by object-oriented languages. But also other (noncode) languages align well with them [8, 14]. Languages that do not satisfy these constraints do not provide sufficient structural information for a composition by superimposition. However, they may be enriched by providing an overlaying module structure [14].

4 Implementation

We have a tool, called FSTComposer, that implements superimposition based on the FST model. Currently, it supports the composition of components written in Java, Jak, XML, and plain text.

FSTComposer expects a list of software components that participate in a composition. It takes a file as input that contains a list of the component names. Then, FSTComposer looks up the locations of the components in the file system.

In FSTComposer, software components are represented by *containment hierarchies* [8]. A containment hierarchy is a file system directory that contains all artifacts (code and noncode) that belong to a component; the directory may contain subdirectories denoting Java packages, etc.

Figure 7 shows the components EmptyCheck and BasicStack containing source and nonsource code artifacts. The composition 'EmptyCheck • BasicStack = CompStack$_2$' composes both their containment hierarchies recursively. For example, the resulting artifact `Stack.java` is composed of its counterparts in EmptyCheck and in BasicStack, matched by name and type.

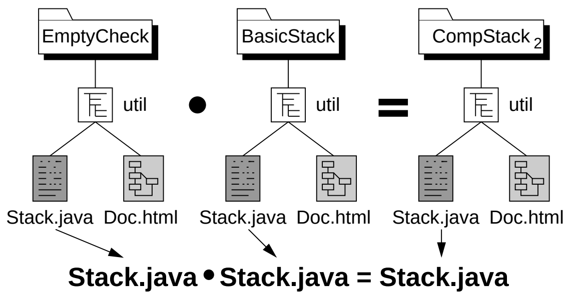

Fig. 7. Composing two containment hierarchies.

Based on an input list of components (essentially, the paths of the containment hierarchies), FSTComposer generates an FST per component. There must be a distinct parser per language. That is, when composing components that contain Java and XML artifacts, two different parsers create the corresponding FSTs.

Currently, our Java and Jak parsers generate FSTs containing nodes for packages, classes, interfaces, methods, fields, imports, modifier lists, and `implements`, `extends`, and `throws` clauses. Packages, classes, and interfaces the nonterminal nodes of a Java FST. The rest are terminals. We have implemented the seven composition rules for terminal nodes, that we have explained in Section 3.2, for Java and for Jak.

Furthermore, we have an XML parser that generates, for each tag, attribute and piece of raw text content, a distinct node; tags become nonterminals; attributes and pieces of text content become terminals; attributes are composed like fields in Java (cf. Sec. 3.2) and pieces of raw text are composed by concatenating their content.

Finally, the text parser is trivial in that it creates nonterminal nodes for directories and simply stores the content of text files in a terminal node each; text nodes are composed by concatenation.

Usually, after the composition step, FSTComposer writes out the composed artifacts. But it can also write out the FSTs of the input and output components in the form of an XML document (containing all information about the Java, Jak, XML, or text artifacts). This language-independent program representation can be the input for further pre- or post-processing of components and component compositions, e.g., optimization, visualization, interaction analysis, or error checking on the basis of FSTs.

The FSTComposer tool along with some examples and case studies can be downloaded from the FSTComposer Web site.[6]

5 Case Studies

We have conducted three case studies to demonstrate the practicality of our approach. Firstly, we have composed a graphical programming tool, called *GUIDSL*, out of a set of software components, which has been implemented by Batory [29]. Secondly, we have composed a series of programs of a small library of graph algorithms, called *graph product line (GPL)*, which has implemented by Lopez-Herrejon and Batory [30]. Thirdly, we have composed several variants of a graphical UML editor, which is an open source program that has been refactored into components by a student. The source code of the three case studies can be downloaded at the FSTComposer Web site.

5.1 GUIDSL

GUIDSL is a tool for software product line configuration [29]. GUIDSL consists of 26 components. For example, there are components that implement the graphical user interface, a parser for grammars that define valid configurations, user event handling, etc. Overall, the code base of GUIDSL contains 294 classes (from which 145 result classes are being composed), implemented by 9,345 lines of Jak code.

GUIDSL was developed in a stepwise manner using components in order to foster extensibility and maintainability. Basically, there is only one valid configuration that forms a meaningful working tool. Other configurations may be valid (syntactically correct) but do not contain all necessary features to work appropriately. We generated a GUIDSL variant consisting of all 26 components, implemented by 7,684 lines of composed Java code.[7]

[6] `http://www.infosun.fim.uni-passau.de/cl/staff/apel/FSTComposer/`

[7] For comparability of the lines-of-code metric, we formatted the code of our case studies using a standard Java pretty printer (`http://uranus.it.swin.edu.au/~jn/java/style.htm`). Furthermore, we counted only lines that contain more than two characters (thus, ignoring lines with just a single bracket) and that are not simply comments (`http://www.csc.calpoly.edu/~jdalbey/SWE/PSP/LOChelp.html`).

We checked the correctness of the composition by testing GUIDSL manually. This was feasible since it is a graphical tool with a fixed set of functions and options that all could be tested. All parser passes and the generation of the composed Java program took less than two seconds.

5.2 Graph Product Line

GPL consists of 26 components written in Jak. For example, the basic components implement weighted, unweighted, directed, and undirected graph structures. Further components implement advanced features such as breadth-first search, depth-first search, cycle checking, the Kruskal algorithm, the Prim algorithm, etc. The overall code base of GPL contains 57 classes (from which 31 result classes are being composed), implemented by 1,308 lines of Jak code.

Beside Jak code, 9 of the 26 GPL components contain an XHTML file that documents the usage and functionality of the graph structures and algorithms. The XHTML files have been prepared by Don Batory and Salvador Trujillo in order to be ready for superimposition [14, 31]. In our case study, we have applied some minor adaptations to match the syntax of FSTComposer, i.e., we have given some XHTML tags unique name attributes in order to specify which tags superimpose other tags. Being superimposed, these XHTML files form the tailored documentation of GPL, depending on the selected components during composition. Due to the lack of space, we refer the reader to the FSTComposer Web site for XHTML examples (of the GPL case study).

Finally, GPL contains some JPEG files that are loaded by the XHTML documentation. During composition, these files are treated like text documents, but their content is not read. A composition of two JPEG files is not necessary. Nevertheless, artifacts with completely opaque content, such as images, align well with the FST model. Artifacts with the same name and type are composed by replacement (a warning is displayed).

Overall, we generated 10 different variants of graph structures along with compatible algorithms with a minimum of 8 and a maximum of 12 components. The code bases of the generated programs range from 200 to 400 lines of composed Java code and 200 to 300 lines of composed XHTML code.

We used GUIDSL to guarantee the validity of the generated configurations [29], e.g., the Kruskal algorithm requires a weighted graph. We checked the correctness of the composed graph implementation with automated tests. The entire composition process, including parsing the Jak and XHTML code, took less than a second per composed program variant.

5.3 Violet

Violet is a graphical UML diagram editor written in Java.[8] It has been refactored by a student as a class project at the University of Texas at Austin.[9] The

[8] `http://sourceforge.net/projects/violet/`

[9] The project was done in the course of the 2006 FOP class at the Department of Computer Sciences of the University of Texas at Austin.

refactored version of Violet consists of 88 components ready for superimposition. They implement support for different UML diagram types as well as drag-and-drop and look-and-feel functionality. Overall, the refactored code base of Violet contains 157 classes (from which 67 final classes are being composed), implemented by 5,220 lines of Java code.

Beside Java code, 83 of the 88 Violet component contain, in summary, 98 property files. A property file contains text-based configuration information of the Violet UML editor, e.g., `edge1.tooltip=Association`. Individual components of Violet provide individual configuration information. Property files are simply composed by text concatenation. There is no further module structure that demands a recursive descent in the FST during composition. As with GPL, Violet contains some JPEG files, but they had not to be composed.

We generated 10 different variants of Violet with a minimum of 51 and a maximum of 88 components. The code bases of the generated programs range from 3,100 to 4,100 lines of composed Java code and 160 lines of text in form of property files.

In order to guarantee their validity, we used the GUIDSL tool for selecting the components of the 10 variants. We tested the variants manually, which was feasible since they differed mainly in their options available in the graphical menus of the editor. All parser passes and the generation of the composed Java and property files took less than two seconds each.

6 Integrating Further Languages

In the previous section, we have illustrated how the FST model abstracts from implementation-specific details of programming languages, while capturing well the abstract hierarchical structure of software components. Currently, FSTComposer supports the composition of components written in Java, Jak, XML, text, and binaries. Due to the generality of the FST model, FSTComposer can be extended to compose also further kinds of artifacts.

Suppose we want to compose software components containing Bali grammar files (a declarative language and tool for processing BNF grammars) [8]. It has been demonstrated that Bali grammars are ready for composition by superimposition. That is, they can be represented as FSTs and composed by superimposition using a proprietary tool [8]. Firstly, we would need a parser that produces FSTs in a format accessible to FSTComposer. Such a parser can be built by extending an existing parser. Secondly, we would have to define the types of nodes (by providing a typically empty subclass per type) that may appear in a Bali FST, e.g., nodes for grammar production rules, axioms, etc. (analogously to nodes for classes and methods in Java). Finally, we would have to define Bali-specific composition rules for composing terminal nodes, e.g., production rules can be extended by providing additional alternatives, similarly to method overriding in Java. Section 7 lists a selection of languages that can be modeled by FSTs.

7 Related Work

Superimposition is a composition technique that has been applied successfully in different areas of software development. Superimposition was initially used for extending distributed programs in multiple places [2, 3]. Subsequently, several researchers adopted this idea in order to merge class hierarchies developed by multiple teams [1], to adapt components [13], to support subject-oriented programming [9, 10], feature-oriented programming [7, 8], and aspect-oriented programming [11, 12], and to implement collaboration-based designs [6]. Several languages support composition by superimposition, e.g., *Scala* [32], *Jiazzi* [25], *Classbox/J* [27], *ContextL* [33], *Jak* [8], and *FeatureC++* [34].

Batory et al. [8], Tarr et al. [10], and Clarke et al. [15] noted that superimposition as a composition technique is not limited to source code artifacts but applies to any kind of artifact relevant in the software development process. Several proprietary tools support the composition of nonsource code artifacts [8, 14, 16–19].

While it has been noted that there is a unique core of all composition mechanisms based on superimposition [8, 10], researchers have not condensed the essence of superimposition into a set of general tools. We believe that our FST model captures the essence of superimposition. It is language-independent. We envision tools that operate on FSTs (or their algebraic representations) to compose, visualize, optimize, and verify software components. Thus, the FST model provides an intermediate format not only for different languages but also for different tools that aim at reasoning about components.

In a parallel line of research, we have developed an algebra and a calculus (incl. operational semantics and type system) of feature composition which is consistent with the FST model [20, 35]. It will allow us to explore general properties of software composition as well as typing issues. Furthermore, it is a means to infer whether a given language fits the FST model and, more interestingly, which properties a language must have to be 'ready' for FST-based superimposition.

Beside superimposition, also other composition techniques have been proposed. For example, composition by quantification, as used in metaprogramming [36] and aspect-oriented programming [37], is a frequently discussed technique. In the context of our FST model, quantification can be modeled as a tree walk [20], in which each node is visited and a predicate specifies whether the node is modified or not. Harrison et al. [23] propose a sophisticated set of rewriting rules that are based on tree walks. Aggregation is another component composition technique. It can be modeled by FSTs that contain nodes that represent themselves components, i.e., that contain FSTs. Even aggregated components can be superimposed, since they have a hierarchical structure that can be represented as an FST. In summary, FSTs are a means to model the connection between different composition techniques and to explore their relationship; FSTs are not specific to superimposition.

So far, we do not consider inter-language interaction. That is, while FST-Composer can compose components containing artifacts written in different languages, it cannot recognize interactions between these artifacts. For example, a Java class may expect some XML document as input, which is defined in another component. Grechanik [38] et al. propose an approach based on recursive types and type reification to bridge the gap between different languages, which can be used in concert with FSTComposer.

Finally, superimposition is a specific instance of model weaving in model-driven development [39] and of graph amalgamation in model theory [40].

8 Conclusion

We model software components by tree structures and component 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.

As a proof of concept, we have developed a tool that implements FST superimposition. Currently, we have parsers for Java, Jak, XML, text, and binaries that generate FSTs ready for composition. Beside generating code for feature composition, FSTComposer is able to generate XML documents representing the FSTs involved in a composition, ready for further processing.

Three case studies have demonstrated the applicability of our approach and our tool: FST superimposition scales to medium-sized programs (10 KLOC). Scalability to larger programs remains to be shown in further work.

We intend to plug various other languages into the tool in order to demonstrate the generality of our approach. C# and Bali have been shown to be compatible with the FST model. Furthermore, we are working on a formalization of the FST model and further tools that operate on FSTs, e.g., a tool that visualizes FSTs and a tool that analyzes interactions between components.

Acknowledgments

We thank Don Batory and Christian Kästner for helpful comments on earlier drafts of this paper, Sebastian Scharinger for implementing the Java and Jak parsers of FSTComposer, Don Batory for releasing the source code of GPL and GUIDSL, and Abhinay Kampasi for refactoring Violet into features.

References

1. Ossher, H., Harrison, W.: Combination of Inheritance Hierarchies. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1992) 25–40
2. Katz, S.: A Superimposition Control Construct for Distributed Systems. ACM Trans. Programming Languages and Systems **15** (1993) 337–356
3. Bouge, L., Francez, N.: A Compositional Approach to Superimposition. In: Proc. Int'l. Symp. Principles of Programming Languages, ACM Press (1988) 240–249

4. VanHilst, M., Notkin, D.: Using Role Components in Implement Collaboration-based Designs. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1996) 359–369
5. Reenskaug, T., Andersen, E., Berre, A., Hurlen, A., Landmark, A., Lehne, O., Nordhagen, E., Ness-Ulseth, E., Oftedal, G., Skaar, A., Stenslet, P.: OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. Journal of Object-Oriented Programming **5** (1992) 27–41
6. Smaragdakis, Y., Batory, D.: Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM Trans. Software Engineering and Methodology **11** (2002) 215–255
7. Prehofer, C.: Feature-Oriented Programming: A Fresh Look at Objects. In: Proc. Europ. Conf. Object-Oriented Programming. Volume 1241 of LNCS., Springer (1997) 419–443
8. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering **30** (2004) 355–371
9. Harrison, W., Ossher, H.: Subject-Oriented Programming: A Critique of Pure Objects. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1993) 411–428
10. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int'l. Conf. Software Engineering, IEEE CS Press (1999) 107–119
11. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: Proc. Int'l. Conf. Aspect-Oriented Software Development, ACM Press (2003) 90–100
12. McDirmid, S., Hsieh, W.: Aspect-Oriented Programming with Jiazzi. In: Proc. Int'l. Conf. Aspect-Oriented Software Development, ACM Press (2003) 70–79
13. Bosch, J.: Super-Imposition: A Component Adaptation Technique. Information and Software Technology **41** (1999) 257–273
14. Anfurrutia, F., Díaz, O., Trujillo, S.: On Refining XML Artifacts. In: Proc. Int'l. Conf. Web Engineering. Volume 4607 of LNCS., Springer (2007) 473–478
15. Clarke, S., Harrison, W., Ossher, H., Tarr, P.: Subject-Oriented Design: Towards Improved Alignment of Requirements, Design, and Code. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (1999) 325–339
16. Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring Product Lines. In: Proc. Int'l. Conf. Generative Programming and Component Engineering, ACM Press (2006) 201–210
17. Bravenboer, M., Visser, E.: Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation Without Restrictions. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2004) 365–383
18. Czarnecki, K., Antkiewicz, M.: 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., Springer (2005) 422–437
19. Kamina, T., Tamai, T.: Lightweight Scalable Components. In: Proc. Int'l. Conf. Generative Programming and Component Engineering, ACM Press (2007) 145–154
20. Apel, S., Lengauer, C., Batory, D., Möller, B., Kästner, C.: An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, Department of Informatics and Mathematics, University of Passau (2007)
21. Batory, D.: Jakarta Tool Suite (JTS). SIGSOFT Softw. Eng. Notes **25** (2000) 103–104

22. Kästner, C., Apel, S., Kuhlemann, M.: Granularity in Software Product Lines. In: Proc. Int'l. Conf. Software Engineering, ACM Press (2008)
23. Harrison, W., Ossher, H., Tarr, P.: General Composition of Software Artifacts. In: Proc. Int'l. Symp. Software Composition. Volume 4089 of LNCS., Springer (2006) 194–210
24. Hutchins, D.: Eliminating Distinctions of Class: Using Prototypes to Model Virtual Classes. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2006) 1–19
25. McDirmid, S., Flatt, M., Hsieh, W.: Jiazzi: New-Age Components for Old-Fashioned Java. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2001) 211–222
26. Nystrom, N., Chong, S., Myers, A.: Scalable Extensibility via Nested Inheritance. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2004) 99–115
27. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2005) 177–189
28. Cardone, R., Lin, C.: Comparing Frameworks and Layered Refinement. In: Proc. Int'l. Conf. Software Engineering, IEEE CS Press (2001) 285–294
29. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Proc. Int'l. Software Product Line Conf. Volume 3714 of LNCS., Springer (2005) 7–20
30. Lopez-Herrejon, R., Batory, D.: A Standard Problem for Evaluating Product-Line Methodologies. In: Proc. Int'l. Conf. Generative and Component-Based Software Engineering. Volume 2186 of LNCS., Springer (2001) 10–24
31. Trujillo, S., Batory, D., Díaz, O.: Feature Refactoring a Multi-Representation Program into a Product Line. In: Proc. Int'l. Conf. Generative Programming and Component Engineering, ACM Press (2006) 191–200
32. Odersky, M., Zenger, M.: Scalable Component Abstractions. In: Proc. Int'l. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2005) 41–57
33. Costanza, P., Hirschfeld, R., de Meuter, W.: Efficient Layer Activation for Switching Context-Dependent Behavior. In: Proc. Joint Modular Languages Conf. Volume 4228 of LNCS., Springer (2006) 84–103
34. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: 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., Springer (2005) 125–140
35. Apel, S., Hutchins, D.: An Overview of the gDeep Calculus. Technical Report MIP-0712, Department of Informatics and Mathematics, University of Passau (2007)
36. Kiczales, G., Des Rivieres, J.: The Art of the Metaobject Protocol. MIT Press (1991)
37. Masuhara, H., Kiczales, G.: Modeling Crosscutting in Aspect-Oriented Mechanisms. In: Proc. Europ. Conf. Object-Oriented Programming. Volume 2743 of LNCS., Springer (2003) 2–28
38. Grechanik, M., Batory, D., Perry, D.: Design of Large-Scale Polylingual Systems. In: Proc. Int'l. Conf. Software Engineering, IEEE CS Press (2004) 357–366
39. Bézivin, J.: On the Unification Power of Models. Software and Systems Modeling **4** (2005) 171–188
40. Böcker, S., Bryant, D., Dress, A., Steel, M.: Algorithmic Aspects of Tree Amalgamation. J. Algorithms **37** (2000) 522–537