Language-Independent Quantification and Weaving for Feature Composition

Stefan Boxleitner[†], Sven Apel[†], and Christian Kästner[‡]

[†] Department of Informatics and Mathematics, University of Passau, Germany
{apel,boxleitn}@fim.uni-passau.de
[‡] School of Computer Science, University of Magdeburg, Germany
kaestner@iti.cs.uni-magdeburg.de

Abstract. Based on a general model of feature composition, we present a composition language that enables programmers by means of quantification and weaving to formulate extensions to programs written in different languages. We explore the design space of composition languages that rely on quantification and weaving and discuss our choices. We outline a tool that extends an existing infrastructure for feature composition and discuss results of three initial case studies. We found that, due to its language independence, our approach is less powerful than aspect-oriented languages but still usable for many implementation problems.

1 Introduction

A *software product line (SPL)* is a set of related programs tailored for a specific domain or market segment [1]. The programs of an SPL are distinguished in terms of features. A *feature* represents the realization of a requirement and its implementation involves the extension and modification of the program that shall provide the feature. Feature composition is the process of merging the code of different features.

Recently, we have proposed a language-independent model of feature composition [2, 3, 4]. In the model, the structure of a feature's implementation is represented as a tree, called *feature structure tree (FST)*, and the process of feature composition is represented as FST superimposition. An FST captures the essential, hierarchical module structure of a given program or feature; superimposition merges two FSTs by merging their corresponding substructures based on nominal and structural similarities.

While superimposition works well in many situations [5, 6, 7, 8], there are certain situations where a different kind of composition is more appropriate [9, 10]. In such situations, an extension a feature applies to a program is expressed more generically and declaratively than possible with superimposition. *Generic* means that an extension should be applicable to (i.e., reusable with) different programs, and *declarative* means that the extension is specified in terms of *where* and *what* to change instead of *how*.

Work on *adaptive programming (AP)* [11], *strategic programming (SP)* [12], *aspect-oriented programming (AOP)* [13], and *subject-oriented programming (SOP)* [14] incorporates and favors a style of composition that determines first which points (*join points*) in the structure and computation of a program are being extended (*quantification*) and that specifies what extensions are being applied to these points (*weaving*). This style has its roots in term rewriting [15], traversals and visitors [16], and meta-programming [17].

Interestingly, composition by quantification and weaving can be described using the FST model. The nodes of an FST are the join points available for being extended by weaving.[1] In the FST model, the process of quantification is represented by a tree traversal that visits each node in a given FST and that decides which nodes are selected for further processing. Then, in a subsequent weaving phase a rewrite (a.k.a. transformation or advice [13]) is applied to all selected nodes. The kind of rewrite depends on the language and may differ, e.g., adding new structural elements, renaming elements, merging or overriding elements' content. Henceforth, we call a program extension that is applied by a feature via quantification and weaving a *modification*; it is a pair of a traversal specification and a rewrite specification.

While the relationship between superimposition and quantification and weaving has been explored using a feature algebra [3] (see Sec. 5), the question for the concrete syntax and semantics of a modification language for feature composition based on the FST model is unanswered. This question is especially interesting as FSTs are language-independent and the envisioned language should take this into account. Previous work either concentrated on specific target languages or on general issues of traversal and rewrite specifications (see Sec. 5).

We would like to know how far we can go with a language-independent model of modifications. Starting from the FST model, we explore the design space for expressing modifications language-independently. This is an improvement over previous work and allows us to understand and model the essence of composition by quantification and weaving. We present a concrete syntax and semantics by means of examples, we outline a tool that integrates well into an existing infrastructure for feature composition, and we discuss results of three initial case studies. The case studies indicate that, due to its language independence, our approach is less powerful than AOP but still usable enough for a significant body of implementation problems.

2 The FST Model

A feature is implemented by one or more software artifacts, each of which can have an internal structure. We model the structure of a feature as a tree, called *feature structure tree* (*FST*), that organizes the feature's structural elements, e.g., classes, fields, or methods, hierarchically. Figure 1 depicts an excerpt of the Java implementation of a simple graph structure and its representation in the form of an FST. One can think of an FST as a stripped-down abstract syntax tree that contains only the information that is necessary for feature composition.

For example, the FST we use to represent Java code contains nodes that denote packages, classes, interfaces, fields, and methods. It does not contain information about the body of methods, etc. A different granularity would be to represent only packages and classes but not methods or fields as FST nodes.

A name and a type are assigned to each FST node. This prevents the composition of incompatible nodes, e.g., the composition of two classes with different names, or of a field with a method of the same name.

[1] Note that, in AOP, join points include events in the computation of a program; as an FST represents the static program structure, join points are of purely static and structural nature.

```
package gl;
class Edge { Node a, b; ... }
class Node { ... }
class Graph {
  Vector<Node> nodes = new Vector<Node>();
  Vector<Edge> edges = new Vector<Edge>();
  int count() { ... }
}
class ColoredGraph extends Graph { ... }
```

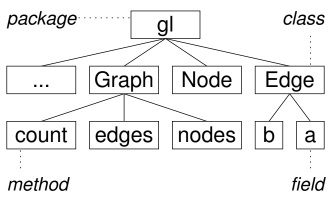

Fig. 1. Implementation and FST of the basic graph library.

Our initial concentration has been on Java artifacts. However, the FST model is applicable to any target language that satisfies two properties [3]: (1) each 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; (2) an element must not contain two or more direct child elements with the same name and type. Many languages have these properties or little effort is necessary to prepare them, e.g., Java, C#, C++, C, JavaCC, Haskell, and UML [5,3,4,7,18,19].

Independently of any particular language, an FST is made up of two different kinds of nodes: non-terminal nodes and terminal nodes. *Non-terminal nodes* are typically the inner nodes of an FST. The sub-tree rooted at a non-terminal reflects the structure of some implementation artifact. The artifact structure is regarded as *transparent* (substructures are represented by child nodes). A non-terminal node has only a name and a type. Examples for non-terminals are packages, namespaces, records, classes, XML elements, lists, and sections.

Terminal nodes are always the leaves of an FST. Conceptually, a terminal node may also be the root of some structure, but this structure is regarded as *opaque* in our model (sub-structures are not represented by child nodes). The content of a terminal is not shown in the FST. A terminal node has a name, a type, and usually some content. Examples for terminals are methods, fields, attributes, properties, initializers, production rules. In order to alter their contents, language-specific rewrites are necessary [2,4]. Here are three examples for Java and similar languages:

– A method can be extended by overriding and calling the original method using the keywords original [6] or Super [5] inside the overriding method body.
– A field declaration can be extended by supplying an initial value and by substituting the field's type with a subtype.
– The list of implemented interfaces of a class can be extended with additional types.

In other languages, such as XML or BNF grammars, similar rules based on overriding, replacement, or concatenation, are useful [5,2,4].

3 Modifications

Conceptually, a modification is a tree traversal that determines a subset of nodes of a given FST and applies rewrites to them. That is, a specification of a modification consists of a *traversal specification* and a *rewrite specification*. In the AOP community, the traversal and rewrite processes are called *quantification* and *weaving* [20]. Since our approach differs in some respects, we chose more general names that match the semantics of the FST model. Modifications are generic in that they can be applied flexibly to different programs resulting in different outcomes. For example, a modification could be used to add a new field to *every* class of a program.

Design Space. What would a language for modifications look like? What properties are mandatory and desirable? First, we have to decide whether the traversal and rewrite specifications are intermixed with code of the target language or whether they are separated from the remaining code. For example, Hyper/J[2] favors a stricter separation than AspectJ[3] or DemeterJ[4] in that there are separate files with composition rules; AspectJ and DemeterJ use first-class language constructs, annotations, or framework code to express the traversal and rewrite specifications. Since we aim at language independence, we separate traversal and rewrite specifications from the target code.

A mandatory property of a traversal specification is that it must be able to select nodes by name and type. Otherwise, a programmer is not able to select anything in the first place, or there is a the risk of ambiguity. Optionally, a traversal language could select a node based on an expression containing patterns and on logical or set-theoretic operators, which increases the expressibility and genericity. This would allow a modification to select multiple nodes at a time, as is possible in languages like AspectJ and Hyper/J.

A rewrite specification either contains the additional code that is injected or added to a target program or it refers to an extra file that contains the code. As in the above case, we choose a strict separation in order to attain language independence. Hence, in our case, a rewrite specification states where the elements/nodes are defined that are being added to or merged with the nodes selected by a traversal. In the case a traversal selects more than one node to be modified, the corresponding rewrite is applied to all target nodes, which is considered a useful mechanism in AspectJ and Hyper/J.

Conceptually, a rewrite may apply any kind of change from adding to deleting and renaming nodes. However, it is reasonable to limit the capabilities for simplicity of the rewrites. For example, a modification that deletes FST nodes or alters node types would be a powerful mechanism – perhaps too powerful. So, for now, we limit the abilities of a rewrite to adding new child nodes and altering the content of nodes via language-specific rewrites. Our algebraic model of features suggests that such a limitation is necessary in order to attain certain desired properties of feature composition, such as associativity [3].

Finally, traversal specifications and rewrite specifications have to be related in order to declare which pairs of them represent modifications. The two specifications could be stored separately, like in Hyper/J, and refer to each other or they could be stored together in a single artifact, like in AspectJ. While the former approach would allow a programmer to combine a traversal specification with different rewrite specifications (e.g., as possible with pointcuts and advice in AspectJ), it imposes an organizational overhead (additional references and files have to be maintained). Since we did not encounter convincing use cases for the former approach yet, we choose the latter, simpler approach.

[2] `http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm`
[3] `http://www.eclipse.org/aspectj/`
[4] `http://www.ccs.neu.edu/research/demeter/DemeterJava/`

Traversal Specifications. A traversal is a function that receives an FST, visits each node, and returns the set of nodes being affected by the modification. Consequently, a language for traversals must take the hierarchical structure of FSTs into account, e.g., in order to express parent-child relations. We illustrate the capabilities of a language that implements our choices step by step by means of the graph example (Fig. 1).

In order to select a node, a traversal expression specifies the node's path starting from the root where each transition from parent to child node is represented by a dot ('.'). For example, selecting the node representing class Graph we would use the traversal expression "gl.Graph" since Graph is a sub-element (child) of gl. Like in AP, AOP, and SOP, wildcards can be used to select multiple nodes. For example, we can select all members of Graph with the expression "gl.Graph.∗", all elements of the package gl whose names end with 'Graph' using the expression "gl.∗Graph", or all elements of any element of gl whose name is 'print' using the expression "gl.∗.print".

Combining two sets of nodes can be expressed by the operation '+'. For example, the expression "gl.∗Graph + gl.∗Edge" selects all classes whose names end with 'Graph' or 'Edge', assuming that there are classes like WeightedGraph and so on. Subtracting two sets of nodes is denoted by '-'. For example, the expression "gl.∗Graph - gl.Color∗" selects all classes whose names end with 'Graph' and do not begin with 'Colored', assuming that there are classes like ColoredGraph and so on. We use operations on sets of join points instead of logical operators on propositions because it fits more the nature of traversals defined on top of the FST model.

The operator '..' is used to express that every direct or indirect child node is selected. For example, "gl..∗" selects any element of the package gl (including sub-packages, classes, methods), "..Graph" selects any class Graph (including top-level and inner classes), and "gl..Graph" selects any class Graph contained in the package gl and its sub-packages.

Expressing that we do not want to select a certain set of join points we use a combination of wildcards and set subtraction. For example, selecting all elements of a program whose names do not end with 'Graph' is expressed with "..∗ - ..∗Graph"

Furthermore, we introduce the operator ':' that is used to specify the desired type of the node to be selected. For example, we can select all inner classes – assuming there are inner classes – of the class Graph by "gl.Graph.∗ : JavaClass". Here 'JavaClass' is the desired type. Of course, it depends on the target language which types may occur in a traversal specification. For example, we can also select all members of all (inner) classes that are not fields using the expression "gl..∗ - gl..∗ : JavaField".

Moreover, we can use wildcards to quantify over types. For example, the expression "gl..∗ : Java∗" selects all nodes whose type name begins with 'Java', in our example, JavaClass , JavaField, and so on. This is useful when features contain artifacts written in different languages. One can omit the type information meaning that the node may have any type.

Rewrite Specifications. A rewrite is applied to a set of nodes selected by a corresponding traversal specification. In the FST model, a rewrite can apply two kinds of changes: (1) add a new child node to a non-terminal, (2) alter the content of a terminal by means of a language-specific rewrite rule. A rewrite specifies which kind of change is applied and where the nodes are located that are being added or merged with the nodes selected by a traversal. For example, a rewrite that adds a new field lock has to specify where the field is actually defined.

A rewrite specification looks similar to a traversal specification. Suppose we want to synchronize access to all classes in the package gl whose names end with 'Graph' using synchronization facilities of package cp. We need two modifications to achieve this: the first modification consists of a traversal "gl.∗Graph : JavaClass" and a rewrite "cp.Lock.lock : JavaField"; it selects all classes whose names end with 'Graph' and adds a field lock that has been defined in the class cp.Lock; Figure 2 illustrates the interplay between traversal and rewrite specification.

The second modification consists of a traversal "gl.∗Graph.∗ : JavaMethod" and a rewrite "cp.Lock.sync : JavaMethod"; it selects all methods of the classes whose names end with 'Graph' and overrides these methods with instances of the method sync. How methods are technically merged/overridden is described elsewhere [4].

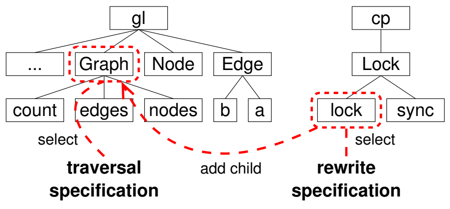

Fig. 2. Interplay between traversal and rewrite specifications.

Of course, modifications are not only useful for extending Java programs. For example, in order to synchronize all C functions in a program we can specify a modification ("..∗ : CFunction", "cp.sync").

Tool Support. We have implemented support for modifications on top of FEATUREHOUSE [4]. FEATUREHOUSE is a tool suite that composes software artifacts by superimposing their FSTs. For every target language supported by FEATUREHOUSE a dedicated parser generates corresponding FST from a given artifact.

Modifications are implemented as described in the previous section. Technically, traversal and rewrite expressions are embedded in an XML document. Traversals and rewrites are straightforwardly implemented on top of FEATUREHOUSE's FST classes using visitors and pattern matching. When applying a modification, FEATUREHOUSE expects code artifacts that are being traversed and used for rewriting in the search path. In our graph example, FEATUREHOUSE would expect files for the classes Graph, Edge, Node, etc. located in the package gl and a file for the class Lock that contains the field lock and the method sync located in the package cp.

Presently, FEATUREHOUSE supports the target languages Java, C#, C, Haskell, Scheme, XML, JavaCC, and UML. It has been shown that further languages can be integrated into FEATUREHOUSE almost solely on the basis of the languages' grammars [4].

Three Case Studies. In three case studies we have implemented modifications for four software systems of different sizes written in two different languages:

	CLA	LOC	TYP	MOD	Description
GPLJ	14	2,439	Java	12	Graph Product Line (Java Version)
GPLC#	14	2,148	C#	12	Graph Product Line (C# Version)
Berkeley DB	300	58,030	Java	4	Oracle's Embedded Storage Engine
AJHotDraw	290	43,368	Java	181	GUI Framework

CLA: # classes; LOC: # lines of code; TYP: types of artifacts; MOD: # modifications

The source code of all software systems of our study can be downloaded from the Web.[5]

In a first study, we have implemented several modifications that add new features to a graph library, called Graph Product Line (GPL). GPL was proposed to serve as an evaluation suite for product line methodologies [21] and so it is a good candidate to test our approach. In order to demonstrate the language independence of modifications, we have implemented five new features with, in summary, twelve modifications on top of a Java and a C# implementation of GPL. The features implement different algorithms for coloring the vertices and edges of a graph and calculating the degree of the graph. For example, a modification injects a field that stores the color value into the classes that represent vertices and edges, and another two modifications inject corresponding setter and getter methods. All three modifications use wildcards in order to define the set of affected classes. A fourth modification changes the method for displaying graphs via overriding in order to adjust the display color according to the vertices and edges being displayed. An observation was that, since the FSTs generated by FEATUREHOUSE differ for Java and C# programs, we were not able to use the same traversal specifications for the Java and C# version of GPL. Basically, we had to change the type annotations of the FST nodes to be selected by the modifications. Apart from this problem, the implementation of the modifications of GPL was straightforward.

In a second study, we have implemented a tracing feature for Berkeley DB (Java Edition). Berkeley DB is an embedded storage engine by Oracle and used in many commercial applications.[6] The tracing feature consists of four modifications. A first modification adds a new class for processing trace information to the code base of Berkeley DB. A second modification injects an object for collecting trace information to every class of Berkeley DB, and a third modification injects corresponding getter methods. A fourth modification selects all methods of Berkeley DB and changes their content to pass information to the tracer that was injected by the second modification. The content change is implemented via overriding, i.e., statements are added in front and in the end of the methods in question. This is similar to AspectJ's around advice for method executions. With 300 classes, the four modifications affect large parts of the code base of Berkeley DB. This indicates the high degree of genericity that can be achieved in the implementation of modifications as well as a certain scalability of our approach and tool, at least for simple modifications.

In a third study, we have explored to what extent it is possible to reimplement the aspects of AJHotDraw[7] with modifications. AJHotDraw is a Java/AspectJ framework for 2D graphics. As modifications in FEATUREHOUSE support only a limited set of changes, we were not able to reimplement all of the 42 aspects. Specifically, we were able to reimplement 23 aspects completely and 13 aspects partially. For readability, we have implemented multiple modifications per aspect and bundled them in directories. For 6 aspects, modification were not expressive enough, in particular, for aspects that advise calls inside method bodies (7 pieces of advice) and that use advanced language constructs such as '`declare error`' or '`cflow`'. Remarkably, most modifications do not affect multiple join points, much like their original pieces of advice. This study

[5] `http://www.fosd.de/fh`
[6] `http://www.oracle.com/technology/products/berkeley-db/je/`
[7] `http://sourceforge.net/projects/ajhotdraw`

indicates that, although modifications are less expressive than aspects, the expressiveness is sufficient for a significant number of aspects in AJHotDraw.

4 Discussion

Our model and implementation of modifications builds on the underlying, language-independent FST model. This implies that modifications are to some extent language-independent as well. Traversal and rewrite specifications rely only on the FST structure of a given software artifact. A drawback of language independence is that there is no semantic information available about the correctness of modifications. There seems to be a trade-off between generality and safety. The aim of our approach is to initiate a discussion about this trade-off. Our model and implementation is on one end of the spectrum between generality and safety. It shows to what extend programs and modifications that features apply can be expressed language-independently.

A further implication of the underlying FST model is that modifications are inherently static transformations. Although motivated by languages like AspectJ, a modification cannot be expressed in terms of events in the computation of a program, which is possible in AspectJ [10]. However, in our case studies we found several situations in which static transformations were useful, and in a recent empirical study we found that use cases for static extensions of programs occur frequently – in contrast to dynamic extensions [22].

Finally, our approach is limited in so far that a modification cannot perform arbitrary rewrites but only add new nodes and change the content of existing nodes (e.g., in order to extend a method body via overriding). Our approach shares this limitation with many other languages for feature composition, e.g., AspectJ, Hyper/J, DemeterJ, and AHEAD [5], which have been nevertheless proven useful in practice.

5 Related Work

Traversals and rewriting have a long tradition in programming [15, 16, 23]. They are the roots of the composition style based on quantification and weaving. With AspectJ this style became popular. AspectJ's pointcuts are traversal specifications. AspectJ supports patterns quantifying over elements and logical operations over propositions of join points that are equivalent to our set operations. In contrast to FEATUREHOUSE, AspectJ builds on a dynamic join point model (which is implemented with static code transformations [24]), but is dedicated to Java.

Hyper/J is closer to our approach than AspectJ. Hyper/J separates target code and composition specification. The bracket rule allows to quantify over some kinds of program elements. Hyper/J depends on Java but supports more expressive composition rules than FEATUREHOUSE, e.g., the merging of two elements with different names.

DemeterJ is a language that allows programmers to express some concerns as traversals over object structures. Traversals are implemented using framework functions. In contrast to the FST model, DemeterJ is more dynamic, as traversals operate on object structures. DemeterJ is specific to Java but the general idea of traversals from which DemeterJ arose is very similar to our approach [11]. In some sense, our approach is an implementation of this concept in the context of general feature composition.

Gray and Roychoudhury propose a technique for constructing aspect weavers using program transformation [25]. The idea is to exploit existing tool support for implementing aspects. However, we aim not only at reusing tools but at a general, underlying model for composition by quantification and weaving. Several researchers propose to raise the abstraction level when expressing extensions to a program, e.g., [26,27]. The FST model is such an abstraction that achieves language independence and that is a substrate for expressing different styles composition.

We have developed a feature algebra that formalizes the key ideas of feature composition [3]. FSTs are represented as sums of elemental path expressions. Modification application and composition are modeled with function application and function composition. Modification application distributes over a sum of path expressions. This way, we model the traversal of the nodes of an FST. The feature algebra was a main motivation for exploring the design space of a language for traversal and rewrite specifications.

6 Conclusion

Starting from the language-independent FST model, we have proposed a language for expressing feature composition based on quantification and weaving. We have explored the design space of such a language, explained our choices, and discussed a concrete incarnation of such a language. We have outlined a tool for composition by quantification and weaving applicable to any kind of software artifact that can be represented by an FST (currently, Java, C#, C, Haskell, Scheme, XML, JavaCC artifacts). Three initial case studies have demonstrated that our approach is feasible and scales to medium-sized software project. However, due to the static nature and language independence, modifications of FSTs are less powerful than AOP or SOP languages, but still usable enough. In further work, we will experiment with more expressive rewrites and conduct further case studies, especially with regard to polylingual systems.

Acknowledgments

Apel's work is supported in part by the German Research Foundation (#AP 206/2-1).

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley (2002)
2. Apel, S., Lengauer, C.: Superimposition: A Language-Independent Approach to Software Composition. In: Proc. Int. Symp. Software Composition, Springer-Verlag (2008) 20–35
3. Apel, S., Lengauer, C., Möller, B., Kästner, C.: An Algebra for Features and Feature Composition. In: Proc. Int. Conf. Algebraic Methodology and Software Technology, Springer-Verlag (2008) 36–50
4. Apel, S., Kästner, C., Lengauer, C.: FeatureHouse: Language-Independent, Automatic Software Composition. In: Proc. Int. Conf. Software Engineering, IEEE CS (2009)
5. Batory, D., Sarvela, J., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Trans. Software Engineering **30**(6) (2004) 355–371

6. Bergel, A., Ducasse, S., Nierstrasz, O.: Classbox/J: Controlling the Scope of Change in Java. In: Proc. Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2005) 177–189
7. Apel, S., Leich, T., Rosenmüller, M., Saake, G.: FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In: Proc. Int. Conf. Generative Programming and Component Engineering, Springer-Verlag (2005) 125–140
8. Anfurrutia, F., Díaz, O., Trujillo, S.: On Refining XML Artifacts. In: Proc. of Int. Conf. on Web Engineering, Springer-Verlag (2007) 473–478
9. Mezini, M., Ostermann, K.: Variability Management with Feature-Oriented Programming and Aspects. In: Proc. Int. Symp. Foundations of Software Eng., ACM Press (2004) 127–136
10. Apel, S., Leich, T., Saake, G.: Aspectual Feature Modules. IEEE Trans. Software Engineering **34**(2) (2008) 162–180
11. Lieberherr, K., Patt-Shamir, B., Orleans, D.: Traversals of Object Structures: Specification and Efficient Implementation. ACM Trans. Programming Languages and Systems **26**(2) (2004) 370–412
12. Lämmel, R., Visser, E., Visser, J.: Strategic Programming Meets Adaptive Programming. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2003) 168–177
13. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (1997) 220–242
14. Tarr, P., Ossher, H., Harrison, W., Sutton, Jr., S.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: Proc. Int. Conf. Software Engineering, IEEE CS (1999) 107–119
15. Visser, E., Benaissa, Z., Tolmach, A.: Building Program Optimizers with Rewriting Strategies. In: Proc. Int. Conf. Functional Programming, ACM Press (1998) 13–26
16. Visser, J.: Visitor Combination and Traversal Control. In: Proc. Int. Conf. Object-Oriented Programming, Systems, Languages, and Applications, ACM Press (2001) 270–282
17. Kiczales, G., Des Rivieres, J.: The Art of the Metaobject Protocol. MIT Press (1991)
18. Apel, S., Janda, F., Trujillo, S., Kästner, C.: Model Superimposition in Software Product Lines. In: Proc. Int. Conf. Model Transformation, Springer-Verlag (2009)
19. Apel, S., Kästner, C., Größlinger, A., Lengauer, C.: Feature (De)composition in Functional Programming. In: Proc. Int. Symp. Software Composition, Springer-Verlag (2009)
20. Filman, R., Friedman, D.: Aspect-Oriented Programming Is Quantification and Obliviousness. In: Aspect-Oriented Software Development. Addison-Wesley (2005) 21–35
21. Lopez-Herrejon, R., Batory, D.: A Standard Problem for Evaluating Product-Line Methodologies. In: Proc. Int. Conf. Generative and Component-Based Software Engineering, Springer-Verlag (2001) 10–24
22. Apel, S., Batory, D.: How AspectJ is Used: An Analysis of Eleven AspectJ Programs. Technical Report MIP-0801, Dept. of Informatics and Mathematics, University of Passau (2008)
23. Lämmel, R.: Scrap Your Boilerplate with XPath-like Combinators. In: Proc. Int. Symp. Principles of Programming Languages, ACM Press (2007) 137–142
24. Hilsdale, E., Hugunin, J.: Advice Weaving in AspectJ. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2004) 26–35
25. Gray, J., Roychoudhury, S.: A Technique for Constructing Aspect Weavers using a Program Transformation Engine. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2004) 36–45
26. Gybels, K., Brichau, J.: Arranging Language Features for More Robust Pattern-Based Crosscuts. In: Proc. Int. Conf. Aspect-Oriented Software Development, ACM Press (2003) 60–69
27. Ostermann, K., Mezini, M., Bockisch, C.: Expressive Pointcuts for Increased Modularity. In: Proc. Europ. Conf. Object-Oriented Programming, Springer-Verlag (2005) 214–240