Research Challenges in the Tension Between Features and Services

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

We present a feature-based approach, known from software product lines, to the development of service-oriented architectures. We discuss five benefits of such an approach: improvements in modularity, variability, uniformity, specifiability, and typeability. Subsequently, we review preliminary experiences and results, and propose an agenda for further research in this direction.

Categories and Subject Descriptors: D.2.11 [**Software**]: Software Engineering—*Software Architectures*; D.1.2 [**Software**]: Programming Techniques—*Automatic Programming*

General Terms: Design, Languages, Theory

Keywords: Feature-Oriented Programming, Feature-Oriented Program Synthesis, Service-Oriented Architecture

1. INTRODUCTION

Service-oriented architecture (SOA) is an emerging field of software architecture. With SOA, software is decomposed into services. A *service* provides a well-defined piece of functionality while hiding implementation details behind an interface. A service infrastructure allows programmers to integrate services that are distributed and written in different programming languages.

SOA is an architectural style dedicated not only to large-scale distributed systems but also to structure applications within the scope of a local environment. Recent research has explored problems regarding modularity, variability, and compatibility of services and related concepts [35, 31, 29, 18, 15]. While there are first encouraging results, a multitude of challenges remains.

Starting from a feature-based approach, we develop a scenario that integrates the notions of services and feature-based product lines. A *feature* reflects a stakeholders' requirement, provides a configuration option, and is an increment in functionality. Programs of a product line are distinguished by their features [21]; the implementations of common features are reused in different programs.

There is a lot of similarity between feature-based approaches and service-based approaches to software system construction; exploring their synergy is beneficial. Both aim at structuring complex software systems into manageable pieces. Our claim is that using features and services in concert can solve present modularity, variability, and compatibility problems that neither of them can solve in isolation.

Typically, features crosscut the hierarchical service architecture, which results in a suboptimal system structure that renders the development, maintenance, and evolution difficult [4]. *Feature-oriented programming (FOP)* is capable of modularizing features whose code would otherwise be scattered across multiple services [30]. *Feature-oriented program synthesis (FOPS)* composes tailored services from features based on a user specification [10, 8]. Finally, the formal foundation of feature orientation [24, 2, 6] provides a straightforward way to set up a formal specification and type system for services based on features, not just on interfaces.

While our approach bears the potential of supporting SOA research and development, as we will argue, it poses several research challenges. We develop our idea of feature-based SOA, point to the potential benefits, and propose an agenda for further research in this direction. We begin with a simple example SOA scenario and discuss subsequently implications for the general case.

2. AN EXAMPLE SCENARIO

We begin with a simple warehouse scenario to illustrate our ideas. We focus here on a specific kind of services, namely on *white-box services* that are developed by a single vendor. However, this does not mean that the services may not be distributed, only that the implementation of all services is accessible. In Section 7, we extend this scenario to *black-box services* that are implemented and provided by different vendors and that are plugged together in an ad-hoc manner.

The warehouse consists of seven services that provide the basic functionality for processing a customer's order:

- acquiring of customer requests,
- checking the availability of ordered goods,
- rating the credit worthiness of customers,
- ordering the goods from an inventory,
- shipping the goods to the customer,
- billing,
- checking the payment of customers.

Figure 1 depicts the service architecture of the warehouse scenario. As is common in SOA, each service provides a

SDSOA'08, May 11, 2008, Leipzig, Germany.

well-defined, language-independent interface and may be implemented in a language of choice.

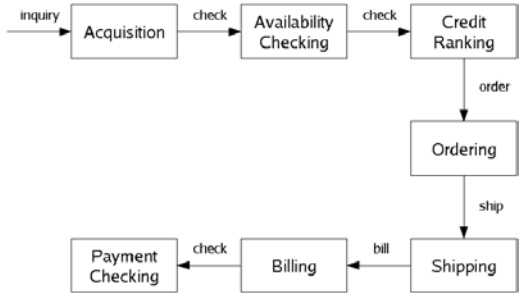

Figure 1: Service architecture of the warehouse.

3. PROBLEM STATEMENT

Although the warehouse scenario is rather small, it is useful to illustrate the problems of suboptimal modularity, variability, and compatibility in SOA. In the basic scenario of Figure 1, the services are well modularized and communicate via interfaces. There are no variability or compatibility issues. But, when exploring the example with a bit more detail, several problems emerge.

Suppose the services of our example are used in different variants of an online warehouse:

Discounting is a variant that offers customers a special discount on some goods for quantities larger than a predefined threshold. The customer either pays less or receives more goods than ordered. This concerns the implementations of *Aquisition*, *Credit Ranking*, *Billing*, and *Payment Checking*.

Status Monitoring is a variant that gives customers the ability to trace the status of the order transaction as well as the logical and physical position of the ordered goods. E-mail notification or a Web-based portal are possible monitoring facilities. This concerns the implementations of all services of the warehouse scenario.

Overseas Orders is a variant that considers the specifics of foreign currency, tolls, logistics, laws, etc. for shipping goods to overseas countries. This concerns the implementations of *Credit Ranking*, *Shipping*, *Billing*, and *Payment Checking* of goods.

A first observation is that the differences between two variants concern multiple (though not all) services of the architecture. For example, to implement the discounting variant, in which customers are billed less, we would have to change the services *Acquisition*, *Credit Ranking*, *Billing*, and *Payment Checking*. Typically, deriving a variant from a service architecture changes a service's implementation and interface only marginally. The core of an individual service and the overall architecture remains untouched. For example, the discounting variant extends the *Acquisition* service's interface (and implementation) to allow customers to query the discount rate. There might even be changes a feature requires that concern only the implementation, but that are not manifest in the interface. For example, the discount variant does not affect the *Billing* service's interface, but only alters the price calculation in the service's implementation.

A straightforward desire is to benefit from the commonality of all variants of a service architecture in terms of reusing assets (such as code) and factoring out the variabilities in a modular fashion. Moreover, a service that is used in different variations of a scenario should be easy to adapt. Otherwise, code replication and an increase in the effort of development along with a decrease in the productivity (due to the difficulties to derive variants) can be expected [17, 16]. Implementing services that provide a superset of the functionalities of all their variants is not a desirable option. Eventually, this will result in bloated and unmaintainable code and make the SOA goal of structuring software appropriately unattainable.

Furthermore, managing the different variants of a SOA scenario is problematic. For example, when we lookup and integrate a service, how do we know to which variants of the scenario it belongs? Which variants of a service are compatible with another service, which may come in different variants, too? FOP and FOPS, two techniques known from the field of software product lines, have the potential solving these problems.

Note that services might be reused in completely different application scenarios, not only in variants of a single scenario. However, we focus on variants of a single scenario. This is reasonable insofar as the services and the architecture is only slightly different in each variant, which is similar to feature-based software product lines [21, 17, 10]. This similarity is the motivation for our approach that integrates features and services.

4. FEATURES AND SERVICES

The different variants of our scenario can be described by and distinguished in terms of features. A *feature* reflects a stakeholders' requirement and provides a configuration option. Of course, in order to provide a feature, the implementation of (some) services has to be extended, i.e., a feature is an increment in service functionality.

Two variants of a scenario differ in their sets of supported features. In our examples, the variants differ only in one feature, i.e., the discounting variant is similar to the base warehouse scenario except that it additionally provides the DISCOUNTING feature.

Figure 2 depicts the warehouse service architecture including the DISCOUNTING feature. The services that are affected by the DISCOUNTING feature are highlighted by arrows. The *Acquisition* service now additionally provides information about the discount (change of interface and implementation), and in *Credit Ranking*, *Billing*, and *Payment Checking* the price calculation is each adapted to grant the discount (change of implementation).

Figure 2 illustrates the following facts of the relationship between features and services:

- A service provides a basic functionality (i.e., a base feature) and a (possibly empty) set of features that extend the base functionality.
- A feature may affect the implementations and the interfaces of multiple services.

The introduction of the notion of a feature to SOA allows us to apply techniques and methods from FOP and FOPS. The idea of FOP is to encapsulate the code of a feature cohesively into a *feature module* [4]. Feature composition joins a service's base code and the code of its features. Figure 3 depicts the warehouse service architecture with a separate

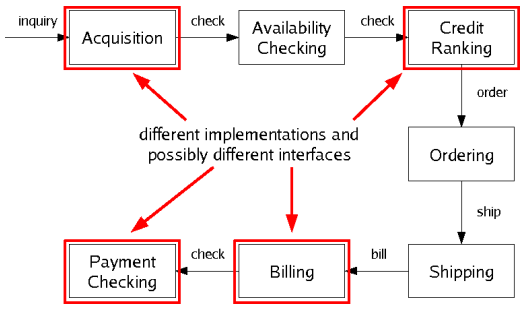

Figure 2: Effects of the discounting feature on the services of the basic warehouse scenario.

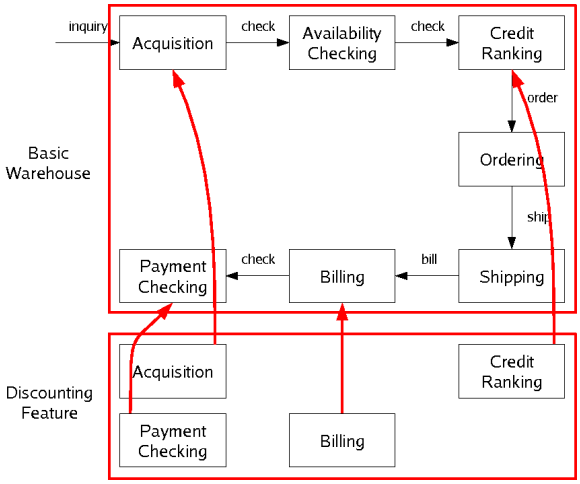

Figure 3: Separating the discounting feature from the basic warehouse architecture.

feature module for DISCOUNTING. Fat (red) boxes represent feature modules and arrows represent extensions to services.

Feature modularization and composition requires a programming language to have a certain expressibility. Several mainstream languages have been extended to support features, e.g., Java, C++, and XML [10, 3, 1]. For example, an extension a feature makes to a service's Java implementation is shown in Figure 4. The class `Bill` of the basic warehouse service *Billing* is refined by the feature DISCOUNTING, denoted by '`refines`'. It overrides the method `getPrice` and adds the method `qualifiesForDiscount` in order to decrease the calculated price of qualified orders by 20 %.

Interestingly, there are FOP extensions of XML [5, 1] that allow programmers to extend a service's interface that is written in WSDL, as illustrated in Figure 5. When composing the base warehouse implementation with the DISCOUNTING feature, the interface of the service *Acquisition* is superimposed with an extension of the same name. DISCOUNTING adds a new operation `discountResponse` allowing a customer to query the discount rate.

FOPS performs feature composition based on a declarative specification [10]. Features are represented by functions that extend a program, in our case, a service architecture including the individual service implementations and interfaces. For example, the feature-oriented model of our warehouse scenario (WH) including the three features DISCOUNTING (DISC), STATUS MONITORING (STAT), and OVERSEA ORDERS (OVER) of Section 3 is modeled by the set:

```
1  class Bill { ...
2    double getPrice(Order o) { ... }
3  }
```

```
4  refines class Bill { ...
5    double getPrice(Order o) {
6      if (qualifiesForDiscount(o))
7        return original(o) * 0.8;
8      else
9        return original(o);
10     }
11   }
12   boolean qualifiesForDiscount(Order o) { ... }
13 }
```

Figure 4: A Java class (top) and a refinement (bottom).

```
1  <definitions name="Acquisition">
2  ...
3    <message name="priceRequest">
4      <part name="orderNumber" type="xsd:int">
5    </message>
6  ...
7  </definitions>
```

```
8  <definitions name="Acquisition">
9  ...
10   <message name="discountResponse">
11     <part name="discountPercentage" type="xsd:float">
12   </message>
13 ...
14 </definitions>
```

Figure 5: A WSDL interface definition (top) and a refinement (bottom).

$$\text{WH} = \{\text{BASE}, \text{DISC}, \text{STAT}, \text{OVER}\} \quad (1)$$

BASE represents the basic service architecture that includes our seven services (Equation 2, names abbreviated). Services, in turn, include a set of code artifacts, not depicted here.

$$\text{BASE} = \{Acqu, Avail, Cred, Ord, Ship, Bill, Pay\} \quad (2)$$

DISC consists of the extensions the discounting feature makes to the warehouse architecture:

$$\text{DISC} = \{Acqu, Cred, Bill, Pay\} \quad (3)$$

This illustrates that the DISCOUNTING feature affects multiple services of the warehouse scenario, i.e., the implementation of the feature is scattered across multiple service implementations. The extensions of DISC are matched (superimposed) with the services of BASE by name. Feature composition scales from composing specific methods or classes, across composing individual services, to composing a whole service architecture.

As features are modeled by functions that modify programs, feature composition ('$\bullet$') is modeled by function composition. The composition of the basic warehouse architecture with the discounting feature is written:

$$\text{WH}_1 = \text{DISC} \bullet \text{BASE} \quad (4)$$

Using such feature expressions, several variants of the warehouse scenario can be described, including those, in which different features are combined:

$$\begin{array}{lll} WH_0 & = & \textsc{Base} \\ WH_1 & = & \textsc{Disc} \bullet \textsc{Base} \\ WH_2 & = & \textsc{Over} \bullet \textsc{Disc} \bullet \textsc{Base} \\ WH_3 & = & \textsc{Over} \bullet \textsc{Stat} \bullet \textsc{Disc} \bullet \textsc{Base} \\ WH_4 & = & \ldots \end{array} \quad (5)$$

The actual composition of software artifacts based on a feature expression is performed automatically by various generator tools, most notably the AHEAD Tool Suite [10].

5. EXPECTED BENEFITS

There are five benefits of a feature-based approach to SOA.

Modularity

The decomposition of a service architecture and its services into features facilitates a better separation of concerns. Code of a feature is encapsulated in a feature module, even though it affects multiple services (cf. Figure 3). The base implementation of a service is not polluted by feature code, which improves code comprehension and maintenance. FOP has been shown to improve modularity in several case studies for software product lines [11, 14, 9, 33, 23, 4] and might improve modularity also in service architectures and related approaches, where modularity has been observed to be suboptimal [35, 31, 29, 18, 15].

Variability

The separation of base service code and feature code allows a programmer to generate different service variants. Feature composition merges the corresponding code fragments of services and features based on a user specification (cf. Equation 5).

Uniformity

Services may be represented in different languages. This includes the implementation (e.g., in Java or C++) and the interface specification (e.g., in WSDL). Feature composition is language-independent and applicable to any kind of software artifact included in a service [5]. For example, FeatureC++ [3] and Jak [10] are languages for composing features written in C++ and Java, respectively. Xak enables the composition of features written in XML [1], which also includes the service's interface written in WSDL.

Specifiability

Programmers, software architects, and users have to distinguish between different variants of an architecture and its services. Otherwise, the composition of (syntactically or semantically) incompatible services may lead to errors, inconsistent system states, and undesirable program behavior. A purely name-based and/or interface-based specification is not sufficient: two services might have equal names and/or interfaces but provide different features (see Typeability). A combination of a name-based, interface-based, and feature-based specification solves this problem. Features represent common abstractions of a domain and help stakeholders to understand a service's semantics during implementation, integration, and orchestration.

Typeability

Without a feature-based specification, services are typed via their names and interfaces. But, as mentioned, not every feature might be manifest in a service's interface. To this end, two variants of a service might be of the same type although they provide different feature sets. Another service might expect a specific behavior that cannot be expressed by an interface. Consequently, the set of features a service provides has to be taken into account during typing. A type system based on interfaces and features solves compatibility and inconsistency problems. Such a type system is partially nominal and partially structural [26, 2].

6. RESEARCH CHALLENGES

We envision several research challenges that arise from the benefits of the symbiosis of features and services.

Modularity

The phenomenon of crosscutting in service, component, and agent systems challenges modularity and has been observed before. Several approaches, most notably *aspect-oriented programming* [22], have been proposed to solve the modularity problem [35, 31, 29, 18, 15]. FOP is closely related to these approaches [4]. A feature-oriented approach is promising insofar as the theory of features, that is based on algebra [24, 6] and category theory [34], enables the automation of feature and service composition, while providing means for simple and precise specification and typing (cf. Section 5). A challenge is to prove the practicality and scalability of crosscutting mechanisms, such as feature modules, in non-trivial SOA case studies. To the best of our knowledge no such studies have been published.

Variability

An automated management of variants becomes increasingly important as the number of variants grows. In SOA, this is especially challenging as there is typically a multitude of services that come in many different variants. An approach based on features profits from the experiences and tools of the field of software product lines [21, 17, 10, 12]. As with modularity, reasonable case studies have to be conducted. Conversely, SOA may become a real-world scenario for researchers and developers that aim at features and product lines.

Uniformity

As discussed before, services may be implemented in different languages. The whole idea of Web services is based on service virtualization [25]. A language-independent communication infrastructure (protocol, interface description, service lookup, etc.) integrates services that have been implemented and deployed by different vendors. Thus, SOA is an excellent use case for evaluating the genericity of feature composition. Whether this genericity is adequate and scales to heterogeneous, large-scale service architectures remains to be explored.

Specifiability

Variants of a service architecture can be specified via features. A prerequisite is that there must be an agreement on the meaning of features in a domain. In the field of software product lines, feature models and ontologies have

been shown to be useful [27, 17]. Assuming a correct feature model, service variant specification can be based entirely on features [21, 17, 12, 7], e.g., as illustrated with Equation 5. Of course, an approach that integrates feature-based and interface-based specification is desirable. The challenge is to establish methods for stakeholders to agree on a feature model and to avoid misunderstandings and preconceptions when using feature-based specifications. The use of formal specification languages is discussed in Section 7.

Typeability

In a type system for SOA, each service has a type. The type is defined by the service's interface and by the set of features it provides. To this end, two services are of the same type if they provide the same interface and the same set of features composed in the same order. That is, the type system is partially structural (e.g., based on the feature structure) and partially nominal (e.g., based on feature names). This notion of service type allows us to define a subtyping relation ('$\leq$'). A service A is a subtype of a service B if A's interface is a subtype of B's interface and A provides a superset of the features of B composed in the same order. The relation $\leq$ can be defined as follows, where $F_n..F_i$ denotes a sequence of features and $\supseteq$ denotes the supersequence relation:

$$\begin{gathered}(\forall A : F \bullet A \leq A) \\ (\forall A, B : A \leq B \Longrightarrow F \bullet A \leq F \bullet B) \\ (\forall A : F_n \bullet \ldots \bullet F_1 \bullet A \leq G_m \bullet \ldots \bullet G_1 \bullet A) \\ \text{iff } F_n..F_1 \supseteq G_m..G_1\end{gathered} \tag{6}$$

A challenge is the integration of a feature-based type system [32, 2] into a formal model of services [13, 28].

7. BLACK-BOX SERVICES

So far, in our discussion we assumed that all service implementations of an architecture are accessible. This makes it possible to implement crosscutting features by means of feature modules that extend the basic services' implementations. This assumption is not unrealistic since SOA is often used in-house as an architectural style for the development of well-structured software systems. However, an advantage of SOA is that services may be black boxes implemented and deployed by different vendors. Software is generated by integrating off-the-shelf services located at different places.

Considering this black-box scenario, the benefits of a feature-based approach are reduced and enhanced at the same time. A feature cannot be condensed into a single feature module anymore. The reason is that service implementations are black boxes and, typically, the vendors do not share code; only interface descriptions are available. This also hinders service composition based on feature composition. Nevertheless, in the scope of a single service provider (i.e., a company that uses SOA to implement large-scale applications) feature composition works as explained; it fails only for services that share the same features but that are implemented by different vendors.

While the benefits of feature modularization and composition are limited to local vendors, the benefits of a feature-based specification and type system increase at the same time. The more vendors contribute services to an architecture, the more a precise, formal agreement on the syntax and semantics of services is necessary. First, there needs to be a common feature model that is well-defined for a domain. Based on this model, vendors can provide a feature-based specification for their services. Beside the name and the relationship to other features (what is required and provided), a specification of a feature defines the semantics expressed in formal specification language, e.g., Alloy [19] or TROLL [20]. The key is that the features can be implemented differently, but they have to satisfy certain constraints. This issue has been explored extensively in the field of program specification. A feature model has to include, for each feature, a language-independent specification.

If there is a common feature model, the feature-based type system plays to its strength. Service integrations can be checked based on their interfaces (interface-based subtyping), their features (feature-based subtyping), and the constraints of the domain (e.g., feature F implies feature G).

8. CONCLUSION

In summary, we see several potential synergies between features and services. A transfer of ideas and experiences would do the field of SOA and FOP(S) good. We have outlined several benefits a symbiosis can bring with it but also several challenges, especially regarding the uniform treatment of services and the formal specification and typing of service compositions. The extended black-box scenario imposes further severe challenges but promises significant benefits of a feature-based approach to SOA.

Acknowledgments

We thank Don Batory and Salvador Trujillo for their helpful comments on earlier drafts of this paper.

9. 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, Dept. Inform. and Math., University of Passau, 2007.

[3] 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.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2), 2008. published online first.

[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, D. Batory, B. Möller, and C. Kästner. An Algebra for Feature-Oriented Software Development. Technical Report MIP-0706, Dept. Inform. and Math., University of Passau, 2007.

[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. From Implementation to Theory in Product Synthesis. In *Proc. Int'l. Symp. Principles of Programming Languages*, pages 135–136. ACM Press, 2007.

[9] D. Batory, C. Johnson, B. MacDonald, and D. v. Heeder. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *ACM Trans. Softw. Eng. Methodol.*, 11(2):191–214, 2002.

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

[11] D. Batory and J. Thomas. P2: A Lightweight DBMS Generator. *J. Intelligent Information Systems*, 9(2):107–123, 1997.

[12] D. Beuche, H. Papajewski, and W. Schröder-Preikschat. Variability Management with Feature Models. *Sci. Comp. Prog.*, 53(3):333–352, 2004.

[13] M. Broy, I. Krüger, and M. Meisinger. A Formal Model of Services. *ACM Trans. Softw. Eng. Methodol.*, 16(1):Article no. 5, 2007.

[14] R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proc. Int'l. Conf. Software Engineering*, pages 285–294. IEEE CS Press, 2001.

[15] A. Charfi, B. Schmeling, and M. Mezini. Transactional BPEL Processes with AO4BPEL Aspects. In *Proc. Europ. Conf. Web Services*, pages 149–158. IEEE CS Press, 2007.

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

[17] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[18] A. Garcia, U. Kulesza, and C. Lucena. Aspectizing Multi-Agent Systems: From Architecture to Implementation. In *Software Engineering for Multi-Agent Systems III*, volume 3390 of *LNCS*, pages 121–143. Springer-Verlag, 2005.

[19] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[20] R. Jungclaus, G. Saake, T. Hartmann, and C. Sernadas. TROLL: A Language for Object-Oriented Specification of Information Systems. *ACM Trans. Inf. Syst.*, 14(2):175–211, 1996.

[21] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[22] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer-Verlag, 1997.

[23] J. Liu, D. Batory, and C. Lengauer. Feature-Oriented Refactoring of Legacy Applications. In *Proc. Int'l. Conf. Software Engineering*, pages 112–121. ACM Press, 2006.

[24] R. Lopez-Herrejon, D. Batory, and C. Lengauer. A Disciplined Approach to Aspect Composition. In *Proc. Int'l. Symp. Partial Evaluation and Semantics-Based Program Manipulation*, pages 68–77. ACM Press, 2006.

[25] A. Nash. Service Virtualization: Key to Managing Change in SOA. *Bitpipe.com*, 2006. White paper.

[26] K. Ostermann. Nominal and Structural Subtyping in Component-Based Programming. *J. Object Technology*, 7(1):121–145, 2008.

[27] X. Peng, W. Zhao, Y. Xue, and Y. Wu. Ontology-Based Feature Modeling and Application-Oriented Tailoring. In *Proc. Int'l. Conf. Software Reuse*, volume 4039 of *LNCS*, pages 87–100. Springer-Verlag, 2006.

[28] M. Perepletchikov, C. Ryan, K. Frampton, and H. Schmidt. A Formal Model of Service-Oriented Design Structure. In *Proc. Austral. Software Engineering Conf.*, pages 71–80. IEEE CS Press, 2007.

[29] A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. In *Proc. Europ. Conf. Object-Oriented Programming*, volume 2743 of *LNCS*, pages 29–54. Springer-Verlag, 2003.

[30] 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.

[31] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: An Aspect-Oriented Approach Tailored for Component-Based Software Development. In *Proc. Int'l. Conf. Aspect-Oriented Software Development*, pages 21–29. ACM Press, 2003.

[32] S. Thaker, D. Batory, D. Kitchin, and W. Cook. Safe Composition of Product Lines. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, pages 95–104. ACM Press, 2007.

[33] S. Trujillo, D. Batory, and O. Díaz. Feature Refactoring a Multi-Representation Program into a Product Line. In *Proc. Int'l. Conf. Generative Programming and Component Engineering*, pages 191–200. ACM Press, 2006.

[34] S. Trujillo, D. Batory, and O. Díaz. Feature Oriented Model Driven Development: A Case Study for Portlets. In *Proc. Int'l. Conf. Software Engineering*, pages 44–53. IEEE CS Press, 2007.

[35] N. Ubayashi and T. Tamai. Separation of Concerns in Mobile Agent Applications. In *Proc. Int'l. Conf. Metalevel Architectures and Separation of Crosscutting Concerns*, volume 2192 of *LNCS*, pages 89–109. Springer-Verlag, 2001.