Technical report

Nr.: FIN-02-2011

Combining Runtime Adaptation and Static Binding in Dynamic Software Product Lines

M. Rosenmüller, N.Siegmund, M. Pukall, S. Apel

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-02-2011

Combining Runtime Adaptation and Static Binding in Dynamic Software Product Lines

M. Rosenmüller, N.Siegmund, M. Pukall, S. Apel

Arbeitsgruppe Datenbanken

Technical report (Internet)
Elektronische Zeitschriftenreihe
der Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg
ISSN 1869-5078

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG)

Herausgeber:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Der Dekan

Verantwortlich für diese Ausgabe:
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Marko Rosenmüller
Postfach 4120
39016 Magdeburg
E-Mail: rosenmue@ovgu.de

http://www.cs.uni-magdeburg.de/Technical_reports.html

Technical report (Internet)
ISSN 1869-5078

Redaktionsschluss: 07.02.2011

Bezug: Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Dekanat

Combining Runtime Adaptation and Static Binding in Dynamic Software Product Lines

Marko Rosenmüller, Norbert Siegmund, Mario Pukall
University of Magdeburg, Germany
{rosenmue,nsiegmun,mpukall}@ovgu.de

Sven Apel
University of Passau, Germany
apel@uni-passau.de

ABSTRACT

Adaptive systems as well as *software product lines (SPLs)* aim at variability to cope with changing requirements. Variability can be described in terms of *features*, which are central for development and configuration of SPLs. In traditional SPLs, features are usually bound statically before runtime. By contrast, *dynamic software product lines (DSPLs)* support feature binding at runtime and can also be considered adaptive systems. DSPLs usually use coarse-grained components for implementation, which limits their customizability. We aim at closely integrating static binding of traditional SPLs and runtime adaptation of DSPLs. We achieve this integration by implementing an SPL using a fine-grained decomposition into features and statically generating a DSPL that is tailored to an application scenario and the execution environment. The generated DSPL supports self-adaptation based on coarse-grained modules that reduce the negative impact on resource consumption. We propose a feature-based runtime adaptation mechanism that reduces the effort for computing an optimal configuration. With a case study, we demonstrate the applicability of our approach and show that a seamless integration of static binding and runtime adaptations optimizes the adaptation process.

1. INTRODUCTION

Software product line (SPL) engineering aims at variable software by generating a set of tailor-made programs from a common code base (e.g., for different customers or application scenarios) [22]. SPL engineers consider *features* as central elements for configuration because they are implementation independent and provide a direct mapping to user requirements. Stakeholders thus use features to describe commonalities and differences of the programs of an SPL. In traditional SPL engineering, features are *bound* statically. That is, a user selects the desired features and a generator creates the corresponding program that contains exactly the selected features.

In contrast to traditional SPLs, adaptive systems and *dynamic SPLs (DSPLs)* offer variability at runtime to adapt to changing requirements [14]. Approaches for runtime adaptation are often based on components and use the high-level software architecture to describe program adaptations [21]. They allow a programmer to specify adaptation rules for reconfiguring components and thereby abstract from the concrete implementation [12, 10, 16]. Consequently, an adaptive system can also be considered a DSPL [7].

DSPLs aim at integrating concepts of traditional SPLs and adaptive systems [1]. Beside approaches that use architectural models to describe program adaptations, there are also DSPL approaches that support feature-based runtime adaptations. For example, some approaches *feature models* to describe dependencies between features and to reason about runtime variability [8, 15, 17, 28] of DSPLs and adaptive systems. Describing also program adaptations in terms of features abstracts from implementation details, simplifies reconfiguration of running programs, and allows for checking consistency of adaptations [8]. Such feature-based approaches use a mapping of DSPL features to the components that are used for implementation [17, 28]. However, components are usually coarse-grained, which limits customizability and applicability of a DSPL. For example, it is required to customize components for embedded systems to remove unneeded functionality and to tailor the components with respect to the hardware. Using small components for this purpose is usually not an option due to an increasing communication overhead.

To bridge the gap between feature-based variability modeling and component-based runtime adaptation, we integrate traditional SPL engineering with DSPLs. We aim at supporting fine-grained customization as well as feature-based runtime adaptation. We achieve this integration by using features for modeling variability, implementing SPLs, and describing program adaptations:

- We model and implement an SPL based on features as known from *feature-oriented software development (FOSD)* [2].
- We propose to generate a tailor-made DSPL from the feature-oriented implementation of an SPL. The DSPL supports dynamic binding of coarse-grained *compound features* (a.k.a. *dynamic binding units*) [24].
- We propose a feature-based approach for runtime adaptation and self-configuration.

In previous work, we have already presented how to generate binding units that can be composed at runtime. In this paper, we present an extended approach for feature-based runtime adaptation and self-configuration that is based on tailor-made binding units. In contrast to DSPLs that use components for implementation, we generate binding units on demand and automatically derive a mapping from the features of a DSPL to its binding units. Consequently, we are able to use features to describe and validate program adaptations. To achieve self-adaptability, we include a customizable adaptation infrastructure into the generated DSPL. By combining this with static optimization of binding units, we

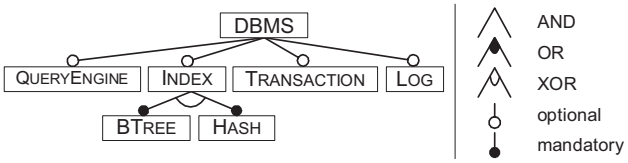

Figure 1: Feature model of a simple DBMS.

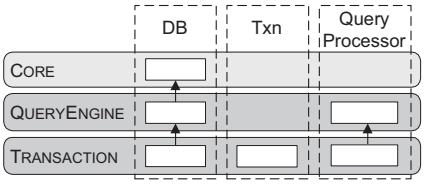

Figure 2: Decomposition of classes (dashed rectangles) along features (horizontal layers). Base classes and class refinements are shown as white boxes.

generate DSPLs with minimal resource requirements. Furthermore, our approach reduces the effort for computing a configuration of a DSPL by minimizing the number of dynamically bound modules. We demonstrate the applicability of our approach with a prototypical implementation of an adaptation framework and a case study. In particular, the contributions of this paper are:

- an integration of static binding as used in SPLs with runtime adaptation as used in DSPLs,
- a feature-based approach for adaptation and self-configuration using binding units,
- a customizable framework for dynamic binding with support for runtime adaptation and validation of configurations using feature models,
- a prototypical implementation and a case study.

In the next section, we present background on feature-oriented software development and generating binding units. In Section 3, we describe our approach for generating DSPLs, which we evaluate in Section 4.

2. COMPOSING FEATURE MODULES

A *feature model* describes valid products of an SPL using a hierarchical representation of features and constraints between them. In Figure 1, we show an example of a feature model for a *database management system (DBMS)*. Mandatory and optional features are denoted by filled and empty dots. To avoid invalid feature combinations, domain engineers use relations between features, such as AND, OR, and XOR, and additional constraints such as *requires* (a feature requires another feature) or *excludes* (two features cannot be used in combination). In general, arbitrary propositional formulas can be used as constraints.

Feature-oriented Programming. Using components to implement variability is sometimes too restrictive because components are coarse-grained, which limits customizability of an SPL. For example, many small features and crosscutting functionality is hard to implement in individual components [13]. In contrast, *feature-oriented programming (FOP)* [23, 5] can be used to implement the features of an SPL in a modular way. FOP thus achieves the same variability in the implementation as it is described by the feature model. For example, we can modularize the transaction management subsystem of a DBMS even though it affects many parts of the whole system. In FOP, features are implemented in *feature modules* as increments in functionality [5]. A user creates a program (a *variant* of an SPL) by selecting a set of features that satisfy her requirements. Based on the feature selection (a.k.a., the *configuration*), a generator composes the corresponding feature modules to yield a concrete program.

A feature module consists of classes and class fragments, as we illustrate in Figure 2 for some DBMS features. The DBMS consists of a CORE implementation and two features QUERYENGINE and TRANSACTION, displayed as layers. The two features cut across the implementation of the classes `DB`, `Txn`, and `QueryProcessor`. In FOP, a programmer thus decomposes a class into smaller class fragments called *base class* and *class refinements* (shown as white boxes) according to the features of a system. A base class implements basic functionality of a class. It is extended by a refinement to contribute to the implementation of a particular feature that cuts across the class. For example, the base implementation of class `DB` is introduced in module CORE and extended in QUERYENGINE and TRANSACTION (denoted with arrows) to implement the according functionality required for these two features.

FeatureC++. FeatureC++[1] is a language extension of C++ that supports FOP [3]. In Figure 3, we depict the FeatureC++ code of class `DB` (cf. Fig. 2). Method `Put` stores data provided as key-value pairs. The refinement in feature QUERYENGINE (Lines 4–9) adds a new field `queryProc` and a new method `ProcessQuery` for processing SQL queries. Feature TRANSACTION overrides method `Put` (Line 12) and invokes the refined method using the keyword `super` (Line 14). Based on the implementation shown in Figure 2, we can generate four different DBMS variants by composing different sets of feature modules: We can generate a simple DBMS consisting only of CORE, but we can also derive variants including the features QUERYENGINE or TRANSACTION or both features.

Static and Dynamic Feature Binding. FeatureC++ supports static binding of features at compile time and dynamic binding at load time or runtime [24]. When binding all features statically, a single binary is generated for the entire program, which is the usual case in traditional SPL engineering. At the class level, the FeatureC++ compiler merges the code of a base class and the refinements of selected features into a single class. For example, static composition of the CORE implementation and feature TRANSACTION of Figure 3, means to generate a single class `DB` that includes all code of the selected features.

For dynamic binding, FeatureC++ generates multiple larger compound features, called *dynamic binding units*. A dynamic binding unit is similar to a component but it includes only required functionality. It contains multiple dynamically bound features that are used in a program at the same time. The FeatureC++ compiler merges these features statically into a single binding unit. To yield a concrete program, a dynamic binding unit is bound as a whole with other

[1] http://fosd.de/fcc

CORE implementation

```
1  class DB {
2    bool Put(Key& key, Value& val) { ... }
3  };
```

Feature QUERYENGINE

```
4  refines class DB {
5    QueryProcessor queryProc;
6    bool ProcessQuery(String& query) {
7      return queryProc.Execute(String& query);
8    }
9  };
```

Feature TRANSACTION

```
10 refines class DB {
11   Txn* BeginTransaction() { ... }
12   bool Put(Key& key, Value& val) {
13     ... //transaction-specific code
14     return super::Put(key,val);
15   }
16 };
```

Figure 3: FeatureC++ code of class DB decomposed along the Core implementation and features QueryEngine and Transaction.

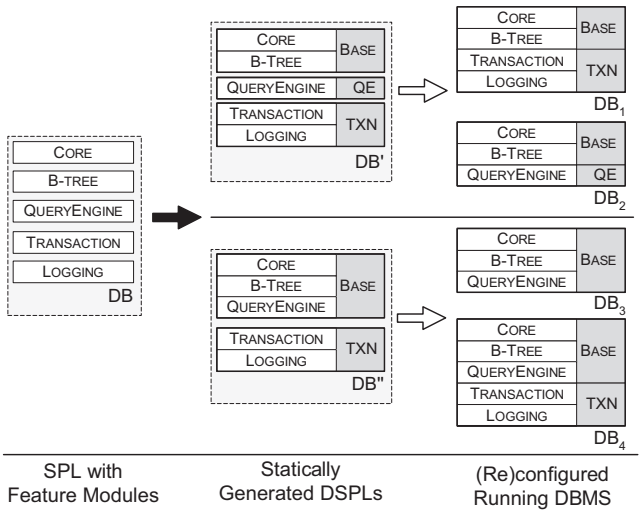

Figure 4: Examples of static composition (➡) resulting in the DSPLs DB' and DB'' and subsequent dynamic composition (⇨) resulting in adaptable running programs DB_1–DB_4.

dynamic binding units at runtime. At the class level, FeatureC++ supports dynamic binding by generating dynamically composable class fragments according to the binding units. For example, to generate a binding unit that contains features QUERYENGINE and TRANSACTION of Figure 3, the code of lines 4-16 is composed into a single class fragment. Multiple class fragments are dynamically composed via delegation using the *decorator* design pattern [11]. This allows us to change the configuration of a class at runtime, which is the basis for *generating* a DSPL from an SPL's source code. For a detailed description of dynamic binding units we refer to [24].

3. GENERATING DYNAMIC SOFTWARE PRODUCT LINES

We generate a DSPL from an SPL by statically selecting the features required for dynamic binding and generating a set of dynamic binding units, as we show in Figure 4. DB' and DB'' are DSPLs generated from SPL DB. The DSPLs consist of multiple dynamic binding units. For example, DB' contains binding units BASE, QE, and TXN. By composing the binding units at runtime, we yield concrete programs DB_1 and DB_2. For that reason, a DSPL contains generated code for dynamic composition of binding units as we describe below. The transformation process from an SPL to a running program can be seen as a *staged configuration* [9]: In a first step, we use FeatureC++ to statically merge multiple features into dynamic binding units (cf. Section 2). In a second step, we compose the generated binding units at runtime. Hence, a DSPL represents a subset of the products of the SPL it was generated from. That is, a DSPL is a *specialization* of an SPL that provides only dynamic variability.

After generating a DSPL, the original features map to the dynamic binding units of the DSPL. For example, features TRANSACTION and LOGGING in Figure 4 map to binding unit TXN in DB'. Features not selected at all (e.g., feature HASH) are not assigned to any binding unit and are thus not present in the generated DSPL. The binding units are used for composition and adaptation at runtime, as we describe next.

3.1 Feature-based Runtime Adaptation

We support feature-based runtime adaptation by describing configuration changes in terms of features and applying these changes at the conceptual level (i.e., using the feature model) before composing the corresponding binding units. At the conceptual level, we represent binding units as *compound features* that are created by merging multiple features of the SPL. In the following, we thus call them *features of the DSPL* and use a feature model to describe dependencies between them. Since the features of an SPL directly represent requirements [2], there is a direct mapping of changing requirements to changes of an SPL's configuration. Usually, there is an n-to-1 mapping because a binding unit contains multiple SPL features. For simplification, we assume a 1-to-1 mapping in the following. Nevertheless, we support arbitrary mappings by transforming the SPL feature model, as we describe at the end of this section.

To support autonomous configuration of DSPLs, we developed *FeatureAce*[2] (Feature Adaptation and Composition framEwork), a customizable framework that supports composition of features at runtime (i.e., dynamic creation of program variants), validation of feature configurations, and also runtime adaptation and self-configuration. To validate configurations at runtime, we integrate the feature model in the form of metadata into the DSPL. Next, we introduce FeatureAce and we then describe how we achieve feature-based adaptation and self-configuration.

A Customizable Adaptation Framework. The FeatureC++ compiler generates a DSPL from an SPL's implementation and FeatureAce, as illustrated in Figure 5. In the resulting executable DSPL, a metaprogram is responsible for autonomous composition and self-adaptation at run-

[2] *FeatureAce* is available online as part of FeatureC++.

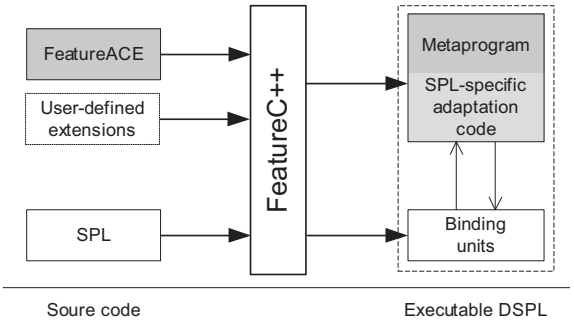

Figure 5: Generating a DSPL from FeatureAce, user-defined extensions of FeatureAce, and an SPL's implementation.

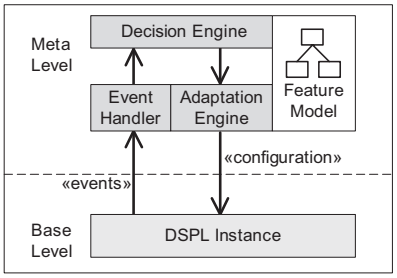

Figure 6: Architecture of a DSPL: Domain code is at the base-level and adaptation code is at the meta-level.

time. The generic metaprogram has access to the base-level (i.e., the binding units) via generated SPL-specific adaptation code.

As shown in Figure 6, FeatureAce provides a decision engine that uses the feature model of the DSPL to reason about validity of changes to the running program. Monitoring code for analyzing the context at runtime is located in the base level. It is developed as part of the SPL since it is usually highly domain specific. For example, code for monitoring DBMS queries can be used to trigger an event for loading a feature that implements a special search index. The monitoring code triggers events that are captured by an event handler, which activates the decision engine. Based on adaptation rules, the decision engine derives a new configuration. The adaptation engine applies configuration changes by loading and unloading binding units.

To further integrate static and dynamic binding, we support static customization of the adaptation infrastructure:

- Monitoring code of the DSPL that triggers adaptation events is implemented in distinct features. Hence, it is possible to use only required monitoring code and to choose between alternative implementations.
- Adaptation rules are also stored in distinct feature modules to allow the programmer to choose only required adaptation rules at deployment time.
- FeatureAce can be customized to choose between manual and autonomous adaptation and to enable validation of adaptations only if required. We thus developed FeatureAce itself as an SPL that can be statically customized.

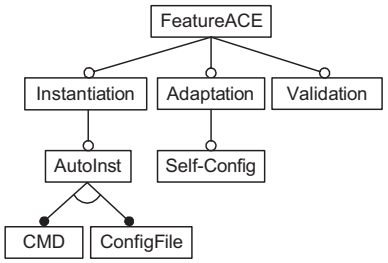

Figure 7: The feature model of FeatureAce. Customization of dynamic product instantiation and runtime adaptation capabilities is achieved by selecting the corresponding features.

Customization of the adaptation infrastructure allows us to cope with changing requirements with respect to the adaptation process (e.g., due to changes in the execution environment). The customizations are usually not needed at runtime and we thus use static binding.

In Figure 7, we show the feature diagram of FeatureAce. Feature AutoInst encapsulates the functionality required for automated SPL instantiation using command line arguments or a configuration file to provide an initial feature selection. Feature Adaptation enables modification of a running SPL instance and feature Self-Config supports rule-based self-configuration. Feature Validation provides functionality to check validity of an SPL variant before composing the binding units. For customization, a user defines the required adaptation facilities of FeatureAce or may add user-defined extensions. FeatureAce extensions (e.g., monitoring code) are implemented as additional feature modules without the need for invasive modifications of FeatureAce. According to the feature selection, the code of FeatureAce extensions and adaptation rules is also statically composed. Hence, only selected adaptation rules and adaptation code are included in a DSPL.

DSPL Instantiation and Adaptation. FeatureAce supports a set of operations for instantiation and adaptation of a DSPL at runtime:

DSPL Instantiation. A DSPL instance is composed from multiple *feature instances*. A feature instance is created by invoking a factory method of FeatureAce that loads the corresponding binding unit (e.g., from a dynamically linked library). Feature instances are composed with each other according to the constraints defined in the feature model. This results in a stack of feature instances that represents a DSPL instance.

DSPL Adaptation. A running DSPL instance can be modified by adding and removing features. For example, DB_1 of Figure 4 can be adapted to DB_2 by removing Txn and adding QE. Features can also be temporarily deactivated and can be reactivated later. This maintains the state of a feature while disabling its functionality.

The described operations are internally used by FeatureAce for runtime adaptation but can also be accessed directly using the API or the network interface of FeatureAce. This

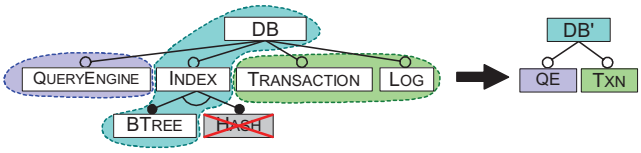

Figure 8: Transformation of a feature model according to defined binding units.

is sufficient when no complex adaptation rules are needed and when events can be directly mapped to a configuration change. For example, when a user selects a menu entry in the running program, the corresponding feature can be activated.

Note that not every feature that can be dynamically composed with other features can also be applied to an already running instance of an SPL. For example, sometimes it is required to initialize the state of the objects of a feature, which cannot always be automated. Hence, some features have to be prepared for runtime adaptation using a special implementation, which is beyond the scope of this paper.

Runtime Validation of Feature Compositions. FeatureAce uses the feature model of the DSPL to validate a feature selection at runtime. To provide only the required variability at runtime, we transform the original SPL feature model according to the generated binding units of the DSPL. In Figure 8, we depict an example for generating the feature model for DSPL DB' (cf. Figure 4). The result is a simplified feature model that corresponds to the generated binding units. FeatureAce uses the DSPL feature model for runtime adaptation to check an adapted configuration against constraints of the feature model (e.g., violation of an XOR constraint). As the feature model and additional boolean constraints (e.g., FEATUREX *requires* FEATUREY) can be transformed into a propositional formula [4], we use a SAT[3] solver to test whether a valid variant can be derived from a feature selection or not. Even though the SAT problem is NP-complete, feature models can be checked efficiently [19].

3.2 Self-Configuration

FeatureAce provides a rule-based mechanism for self-configuration that we describe next.

Configuration Constraints. Our approach for runtime adaptation is based on adaptation rules that describe how a configuration C of an SPL must be changed when an event E is triggered. A configuration C of a program P of a DSPL is a set of features that is included in the program. We derive a configuration C from requirements R that define which features of the DSPL must be included in a valid program. In the simplest case, the requirements are a set of required features (e.g., a user-defined feature selection). In general, however, a requirement may be an arbitrary *configuration constraint* (i.e., a propositional formula over the set of available features) that restricts the set of valid configurations [25]. A configuration constraint is not different from a domain constraint of a feature model but it is added and removed during configuration. For example, to express that a feature must be included in a program we can define

[3]Boolean satisfiability problem.

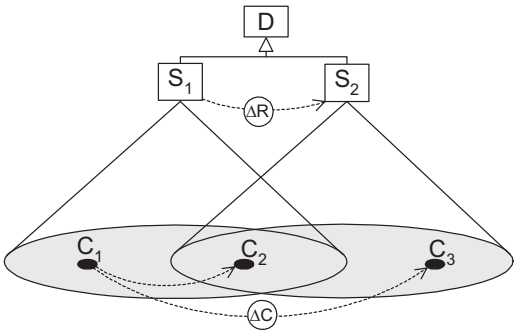

Figure 9: DSPL D with specializations S_1 and S_2 and configurations C_1-C_3.

a *requires* constraint for that feature.

As an example consider the feature diagram of Figure 1 with an initial set of requirements R (e.g., defined by a user):

$$R = \{\text{QUERY}, \text{INDEX}\} \quad (1)$$

R defines that features QUERY and INDEX must be included in a valid configuration. We can thus derive a configuration C that satisfies R:

$$C = \{\text{QUERY}, \text{INDEX}, \text{HASH}\}. \quad (2)$$

Because C must also satisfy the constraints defined in the feature model, such as the XOR-constraint between HASH and BTREE (cf. Fig. 1), it must include one of the two features. In our example, we have chosen feature HASH. While C represents a single configuration, R describes a *specialization* S of our DSPL that represents multiple configurations, as illustrated in Figure 9. DSPL D that has two specializations S_1 and S_2, which we denote with inheritance [25]. Each specialization represents multiple configurations (illustrated with a cone). For example, C_1 and C_2 are configurations of S_1.

We can represent a set of requirements R as a single propositional formula using a conjunction of all requirements. For example, R from equation (1) corresponds to the boolean constraint QUERY $\wedge$ INDEX. Since a feature model can also be translated into a propositional formula [4], we can check whether a configuration C satisfies the requirements R for a feature model FM: If $FM \wedge R$ is `true` for configuration C then C is valid with respect to R. Furthermore, we can use a SAT solver to check whether R is a valid set of requirements with respect to FM. This can be done by checking if we can derive at least a single valid configuration, i.e., $FM \wedge R$ must be satisfiable [27]. This allows us to check at runtime whether a specialization S_i has a valid configuration (cf. Figure 9).

Adaptation Rules. The current configuration C of a running DSPL is modified by a configuration change ΔC (i.e., a reconfiguration) the defines which features are added to C and which features are removed from C during adaptation. However, as we explain below, it is usually too restrictive to directly define configuration changes in an adaptation rule. Instead, an adaptation rule describes changes with respect to the active requirements R of a DSPL. We thus define an adaptation rule A as a pair $(E, \Delta R)$ where E is the event that triggers the rule and ΔR are modifications that must be applied to R when E is triggered. ΔR is a pair $(\Delta R_{\oplus}, \Delta R_{\ominus})$

of added and removed requirements. We use operator $\bullet$ to denote adaptations (i.e., application of ΔR to R):

$$R' := \Delta R \bullet R \quad (3)$$
$$:= (R \setminus \Delta R_{\ominus}) \cup \Delta R_{\oplus}. \quad (4)$$

R' must be satisfied after applying rule A. Hence, ΔR does *not* directly modify the configuration of a DSPL but it modifies a set of requirements R that describe a specialized DSPL. As illustrated in Figure 9, applying ΔR to S_1 results in a different specialization S_2.

From a modified set of requirements R', we derive a modified configuration C'. In Figure 9, we can derive two valid configurations C_2 or C_3 from S_2. For runtime adaptation, we have to choose one of these configurations. For example, we may choose a configuration with the smallest number of features. Finally, we derive the corresponding configuration change ΔC, which is a pair $(\Delta C_{\oplus}, \Delta C_{\ominus})$, from the old configuration C and new configuration C':

$$\Delta C := (\Delta C_{\oplus}, \Delta C_{\ominus}) \quad (5)$$
$$\Delta C_{\oplus} := C' \setminus C \quad (6)$$
$$\Delta C_{\ominus} := C \setminus C' \quad (7)$$

$\Delta C_{\oplus}$ is the set of features that are added to C and $\Delta C_{\ominus}$ are removed from C during adaptation. As a complete example, consider the DBMS from equations (1) and (2) with an adaptation rule A that is triggered on event E_{Range}, meaning that range queries are frequently used:

$$A = (E_{Range}, (\{\text{BTREE}\}, \emptyset)) \quad (8)$$
$$R' = (\{\text{BTREE}\}, \emptyset)) \bullet R \quad (9)$$
$$= \{\text{QUERYENGINE}, \text{INDEX}, \text{BTREE}\} \quad (10)$$
$$C' = \{\text{QUERYENGINE}, \text{INDEX}, \text{BTREE}\} \quad (11)$$
$$\Delta C = (\{\text{BTREE}\}, \{\text{HASH}\}). \quad (12)$$

Rule A adds feature BTREE to R (i.e., BTREE has to occur in a valid configuration), which results in the modified requirement R'. From R' we derive a new configuration C' by adding feature BTREE and removing feature HASH. By contrast, a direct configuration change is too restrictive. For example, an adaptation rule that adds feature BTREE directly to configuration C results in violation of the XOR constraint of the feature model (either BTREE or HASH have to be selected). Furthermore, we cannot define rules to remove features because they may be required by other constraints.

We specify adaptation rules in a declarative language, as shown in the example in Figure 10. The corresponding grammar is shown in Figure 11. A rule consists of a name, a named adaptation event E (e.g., `OnTxn` in Line 2) that triggers execution of a set of actions ΔR, which modify the current configuration of the DSPL. An action can add or remove a named configuration constraints using keywords `addConstraint` and `removeConstraint` followed by a constraint definition (Line 5) or a constraint name respectively (Line 8). Each constraint has a name to be able to remove it from the requirements of a DSPL, as shown in Line 8.

Currently, we define adaptation events in monitoring using the host language. It would also be possible to use a declarative specification as it is done in related approaches [12].

Applying Adaptations. Before computing a new configuration when applying an adaptation rule, FeatureAce checks

```
1 //Load transaction management
2 BeginTxn : OnTxn => addConstraint(TX: Transaction)
3
4 //Process range queries
5 BeginRQ : OnRangeQuery => addConstraint(RQ: Btree)
6
7 //Remove constraint RQ
8 EndRQ : OnRangeQueryEnd => removeConstraint(RQ)
```

Figure 10: Two adaptation rules that add named constraints TX and RQ (Lines 2 and 5) and a rule that removes constraint RQ (Line 8).

```
1  AdaptScript: Rule+ ;
2  Rule: RuleName ":" EventName "=>" Action+ ";" ;
3  RuleName: ID ;
4  EventName: ID ;
5  Action: AddReq | RemoveReq ;
6  AddReq: "addReq" "(" ReqName ":" Constraint ")" ;
7  RemoveReq: "removeReq" "(" ReqName ")" ;
8  ReqName: ID ;
9  Constraint: FeatureName
10           | "!" Constraint
11           | "(" Constraint ")"
12           | Constraint ConstraintOp Constraint ;
13 ConstraintOp: "&&" | "||" | "->" | "<->" ;
14 FeatureName: ID ;
```

Figure 11: Grammar of FeatureAce's adaptation rule specification language.

whether an adaptation is really needed: If a set of requirements R_i represent a specialized DSPL S_i (e.g., S_1 in Figure 9) then $R_{i+1} = \Delta R \bullet R_i$ corresponds to a new specialization S_{i+1} (e.g., S_2 in Figure 9). If S_i and S_{i+1} are overlapping then there are configurations that can be derived from both specializations (e.g., C_2 in Figure 9). Hence, if the old configuration is also a valid configuration of the new specialization, we do not have to adapt the running program. For example, adaptation of S_1 to S_2 in Figure 9 for configuration C_2 does not require a program adaptation. Hence, the decision engine of FeatureAce first checks whether the current configuration C_i satisfies the new requirements R_{i+1}.

If this is not the case, we have to find a new configuration C_{i+1} that satisfies R_{i+1}. To test if there is any valid configuration that satisfies R_{i+1}, the decision engine checks satisfiability of the feature model including the changed requirements. If there are multiple valid configurations, the decision algorithm has to choose the *best* one. Which configuration is the *best* depends on several requirements and is a challenging task in current research [10]. For example, we may choose the configuration with the smallest number of features or the smallest number of required adaptations. Other optimization goals are non-functional requirements such as memory consumption, performance, or quality of service. We currently choose the configuration with the smallest number of configuration changes to reduced the required adaptation changes. For that reason, FeatureAce tries to keep already configured features to minimize changes. Features are removed from a configuration when they violate a constraint. Hence, when an adaptation rule removes a constraint from the requirements R, this does not always cause a configuration change. Furthermore, we removed features that have not been used for a configurable time span to provide a simple mechanism for reducing resource consumption.

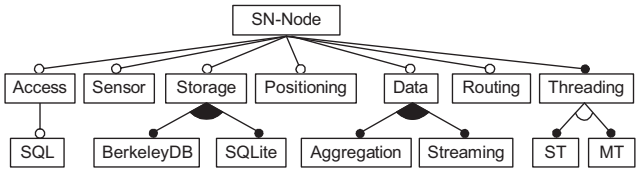

Figure 12: Feature diagram of an SPL for sensor network nodes.

Hardware	Role	Binding Units
Simple	Sensor	StaticSense
Advanced	Positioning	Core, Positioning
	Sensor	Core, Sense
	DataAggregator	Core, QueryProc, Aggregation
	AccessNode	Core, QueryProc, Streaming

Table 1: Examples of different roles for two kinds of devices.

To reduce consumed resources, a configuration can be explicitly minimized by rules, e.g., triggered by low working memory. In future work, we plan to use more sophisticated mechanisms to trigger unloading of features based on non-functional requirements. For example, we may remove unused features based on statistics and the workload of the system or optimize a configuration using CSP[4] solvers [6].

Adaptation Rules for Binding Units. Above we assumed that there is a 1-to-1 mapping of the features of the SPL to the features of the DSPL (i.e., the binding units). In practice, however, there is an n-to-1 mapping of SPL features to DSPL features because multiple features of the SPL are merged into a single binding unit of the DSPL. Hence, when transforming the feature model according to the defined binding units, as shown in Figure 8, we also have to transform the adaptation rules that are defined with respect to the original SPL features. This transformation is easy to achieve by replacing each feature in adaptation rules with the binding unit of the feature.

After transformation, we can check whether all adaptation rules can be applied to the feature model of the DSPL using a SAT solver. A rule is invalid with respect to the chosen binding units if its application to the feature model results in an unsatisfiable formula. For example, features Transaction and Log in Figure 8 are part of the same binding unit. Hence, both features must be present at the same time in the generated DSPL. Consequently, an adaptation rule that requires *either* Transaction *or* Log cannot be used with this DSPL.

4. CASE STUDY AND DISCUSSION

By means of a case study, we demonstrate the flexibility of our approach and show that binding units can reduce the time needed for runtime adaptation. We use a prototypical implementation of FeatureAce for FeatureC++. However, the concept can be applied to other languages as well. As application scenario, we use a sensor network simulation.

4.1 An SPL for Sensor Network Nodes

A sensor network (SN) is a network of interconnected embedded devices (e.g., via radio communication), which sense different kinds of information (temperature, light, etc.) [18]. There can be different kinds of nodes in a network. *Sensor nodes* measure data, store it locally, and send it to other nodes. *Aggregation nodes* aggregate data (e.g., computing the mean value) and data in the network is accessed from the outside via *access nodes*. By using an SPL of node software, we can *generate* different program variants tailored to the different kinds of SN nodes.

[4]Constraint satisfaction problem

In Figure 12, we depict an excerpt of the feature diagram of the SN-Node product line we implemented in FeatureC++. Subfeatures of Data are used for aggregation in data nodes (Aggregation) and streaming in access nodes (Streaming). A node can not only play a single role (e.g., being a sensor node) but multiple roles at the same time. For example, a node may aggregate data but may also be responsible for accessing the network. To compensate node failures and for efficiency, the role may change over the lifetime of a node. For example, if the access node fails due to exhausted battery power, a different node can reconfigure itself to provide this service. Due to hardware constraints, not every physical node can play any role. For example, only a node with sufficient storage capacity can be used for data aggregation.

We define feature configurations for the different roles. Three main decisions influence the configuration process:

1. *Dynamic Binding:* For embedded devices that do not allow dynamic changes to loaded program code (because the executable code is stored in ROM), we do not support runtime adaptation and statically generate program variants.
2. *Runtime adaptation:* For all other nodes we generate a DSPL using a subset of all features. We deploy only the features that are required for the used operating system, the hardware, special customers needs, and the roles a node can play.
3. *Binding units:* To avoid a high overhead at runtime and to reduce the number of possible variants for reconfiguration, we merge features into binding units when they are always used in combination.

4.2 Defining Binding Units

In Table 1, we show a sample assignment of roles for two types of node hardware and the corresponding binding units, which demonstrates the flexibility of our approach. The binding units are composed from the features of Figure 12. We depict sample configurations in Table 2. In our example, simple node hardware (*Simple* in Table 1) with highly constrained resources does not support runtime adaptation and can only be used for sensor nodes. We use a statically composed variant for these nodes (binding unit StaticSense in Table 1). The statically generated program does not include any code for runtime adaptation. It has a binary size of 48 KB, which is only half of the size of a runtime-adaptable variant with the same features (104 KB). This overhead mostly comes from code of the infrastructure for runtime adaptation, which is independent of the number of features. The overhead is quite small compared to larger programs such as a node with stream processing, which has a binary size of 576 KB. Hence, our approach allows us to

Binding Unit	Features
StaticSense	Positioning, Routing, Sensor, Radio, ST
Core	Routing, Radio, Wi-Fi, MT
Positioning	Positioning
Sense	Sensor
QueryProc	Access, SQL, Data, Storage
Aggregation	Aggregation, SQLite
Streaming	Streaming, BerkeleyDB

Table 2: Sample configuration of different binding units.

apply self-configuration also on resource constrained environments. Nevertheless, it still limits the applicability of runtime adaptation and may require static binding when storage capacity is highly limited. With our approach, a user can choose at deployment time whether to use static binding or to support runtime adaptation.

Hardware with less resource constraints (*Advanced* in Table 1) that supports reconfiguration at runtime is used for different roles. An advanced node is deployed with role *Positioning*, for computing the relative position of the node. A node unloads the feature when the position has been determined. If a *Sensor*, a *DataAggregator*, or an *AccessNode* is needed, an advanced node loads the required binding units. The node may also play different roles at the same time. For example, to process a streaming query, a DataAggregator additionally loads the STREAMING binding unit.

We observe that our approach provides high flexibility with respect to possible deployment scenarios. We can define different feature configurations according to the used hardware at deployment time *and* according to required functionality at runtime. For example, a Sensor node uses different binding units but a similar set of features depending on the used hardware (Simple or Advanced). We can also define completely different binding units and feature selections according to available hardware, application scenarios, etc.

4.3 Self-Adaptation

Adaptation Rules. For self-adaptation of the DSPL, we define the adaptation rules within dedicated feature modules of the sensor network. For example, we place rules for activating and deactivating stream processing in feature STREAMING. The rules are included in a running program only if the corresponding feature is selected for dynamic binding. Based on the defined rules, a DSPL autonomously reconfigures itself according to the required features at runtime. In our scenario, a node loads the streaming binding unit when it receives a streaming query.

Reconfiguration of nodes is triggered by events spawned in monitoring code of the DSPL. We implement the monitorings in distinct feature modules that extend classes of the application SPL to separate adaptation code from the SPL's implementation. For example, to activate stream processing, the monitoring code captures incoming queries and triggers the adaptation event when a streaming query is found. The corresponding rule adds a constraint for feature STREAMING (i.e., the feature must be included in a valid configuration). Another rule removes the constraint after all streaming queries have been processed. We do not directly remove the feature which would result in an unneeded reconfiguration when the feature is used again. A feature is only removed when it is excluded by other constraints or when non-functional requirements such as limited working memory force to remove unneeded features. For example, we use a rule to unload the positioning feature when the position of the sensor has been calculated.

Reconfiguration. In Figure 13, we depict evaluation results for the adaptation process.[5] We analyzed the time needed for calculating whether an adaptation is needed and the time for reconfiguration. To show the benefits of statically optimizing the feature model, we compared reconfiguration of a sensor node (1) using the original feature model of the SPL including all 55 features and (2) using the transformed feature model of the DSPL with 6 features (i.e., one feature per binding unit; cf. Sec. 3.1). In the diagram, we depict the time a node requires to process queries that are sent every 300 ms.

Begin of stream processing is triggered by streaming queries (denoted with b in Fig. 13), which results in a runtime adaptation to load binding unit STREAMING. The adaptation must be finished before the query processing can continue. The first streaming query is detected after 5 s. Loading the STREAMING feature takes 20–60 ms (note that we use a logarithmic scale) and increases the response time because the execution is continued after reconfiguration. Calculating the new configuration takes less than 1 ms. Assuming a minimal adaptation time of 20 ms, a node cannot reconfigure itself more than 50 times per second.

End of stream processing is denoted with (e). Instead of unloading the Streaming feature, a rule removes the constraint added before. Hence, all following adaptation events do not cause a reconfiguration. Nevertheless, the adaptation events increase the response time by about 0.3 ms when using the complete feature model with 55 features and 0.05 ms for the DSPL model with 6 features. This computation time is required for checking whether the node has to be reconfigured due to the context change. We use a SAT solver for this purpose. Compared to a reconfiguration that takes 20-50 ms, 0.3 ms is a very small overhead. However, it means that the node cannot handle more than about 3000 changes of the adaptation context per second even though no adaptation is needed. On an embedded device this would be much less due to limited computing power. By contrast, the node with the simplified feature model with 6 features requires only 0.05 ms for checking whether an adaptation is needed. This demonstrates the importance of reducing the variability for runtime adaptation by optimizing the feature model.

4.4 Conclusion

In our case study, we combined static feature binding with support for feature-based runtime adaptation. We have shown that we can achieve autonomous reconfiguration by including the adaptation mechanism and the feature model into the running DSPL. By generating binding units, we can further optimize runtime adaptations, as we discuss next.

Implementation-independent Adaptations. Using features, we can provide adaptation mechanisms that are independent of the modules actually used for dynamic binding. Hence, we can generate binding units that fit to an

[5]For evaluation, we used an AMD 2.0 GHz CPU and Windows XP.

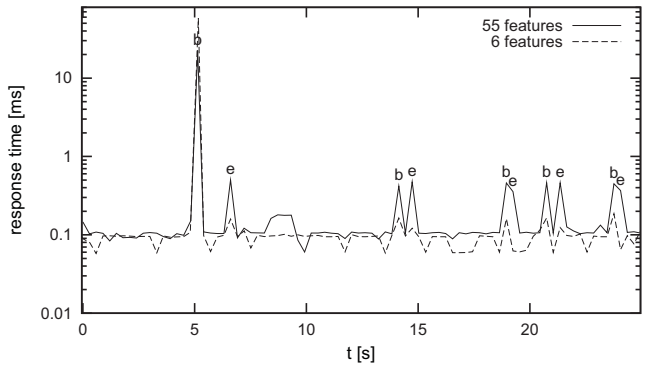

Figure 13: Response time (log scale) during reconfiguration of a query processing sensor network nodes using a feature model with 55 features and a simplified model with 6 features. Begin and end of stream processing are denoted with (b) and (e).

application scenario and the hardware while being able to reuse adaptation rules. When generating binding units, we transform the adaptation rules accordingly.

Composition Safety. Using a feature model, we ensure that adaptations are correct with respect to domain constraints. As we have shown, this can be efficiently done at runtime before creating a variant by using a SAT solver. Furthermore, we can check if an adaptation rule is valid with respect to the feature model of the DSPL before runtime.

Resource Consumption. We provide an adaptation mechanism with low resource requirements (e.g., binary size, computing power) due to (1) customization of the adaptation infrastructure and (2) customization of binding units by removing unused code. The flexible size of binding units minimizes dynamic binding and enables static optimizations, as we have analyzed in previous work [24].

Computational Complexity. We have shown that we can reduce computations at runtime in two ways: (1) by avoiding unneeded adaptations and (2) by reducing the computations for checking satisfiability by transforming the feature model according to actually available variability. The time required for computing a valid configuration is small compared to an actual reconfiguration even when using a feature model with 55 features. However, frequently checking whether a reconfiguration is needed can easily require more computing power than available. Hence, it is important to reduce the computation time by optimizing the feature model.

Including also non-functional requirements in these computations is a challenging task [10]. However, our approach reduces the overall computational complexity and can be combined with CSP solvers to consider also numerical constraints (e.g., limited memory consumption) when computing an optimal configuration at runtime [26].

Constraint-based Adaptations. Directly modifying the configuration or an architectural model can result in unneeded reconfigurations and may cause configuration conflicts. Instead, we compute a new configuration based on the current configuration of a program and a set of requirements the new configuration has to fulfill. This avoids configuration conflicts and frequent reconfigurations, as we have shown our evaluation.

5. RELATED WORK

There are several approaches that use components and an architecture-based runtime adaptation as proposed by Oreizy et al. [21]. We further abstract from implementation details and use features for configuring a program at runtime. This allows us to reason about configuration changes at runtime on a conceptual level and to describe adaptation rules in a declarative way without taking the high-level architecture into account.

There are approaches that apply SPL concepts to develop adaptive systems and approaches that use feature-oriented concepts for modeling dynamic variability in SPLs [10, 15, 17, 28, 8]. We aim at building a foundation for integrating them using features for SPL configuration as well as runtime adaptation. We compare our approach with respect to the most prominent approaches.

Some approaches describe dynamic variability in terms of features [17, 28, 8]. Lee et al. use a feature model to describe the variability (static and dynamic) of an SPL and they suggest to manually develop components (i.e., feature binding units) for implementing dynamic variability [17]. We decide *at deployment time* which binding time to use and use features (i.e., not implementation units) for configuration and program validation at runtime.

Floch and Hallsteinsen et al. present with MADAM an approach for runtime adaptation that uses SPL techniques as well as architectural models [10, 15]. They propose to model variability using component-based SPL techniques [15]. We use features for modeling variability and runtime adaptation to further abstract from the underlying implementation.

Cetina et al. use feature models to describe variability of an adaptive system [8]. To adapt a system, they modify a configuration by adding or removing features. This results in the problems discussed in Section 4.4, which we solve by using constraints to describe current requirements on a system. Furthermore, we seamlessly integrate SPL engineering and runtime adaptation by applying SPL concepts to adaptation code (e.g., adaptation rules) and by supporting static binding of features and merging of features into binding units.

Morin et al. describe variability with a feature model and realize variability of the component model of an adaptive system with aspect-oriented modeling (AOM) [20]. They use aspects to describe model adaptations and reconfigure a program based on changes of the model. In contrast, we operate on features that are not only implementation independent but also independent of the component model of a system. Hence, our approach can be combined with an approach for model adaptation. This allows us to validate a configuration before adapting the component model.

We use feature modules for implementing adaptive systems, but our approach for feature-based adaptation may also be used with other implementation units such as components or aspects [20, 8, 17, 28]. In this case, features are still used to describe adaptation changes. After a new configuration has been derived and validated, a corresponding set of components has to be determined. Some of the approaches above provide advanced capabilities for runtime adaptation not considered here (e.g., adaptation planning, state transfer, etc.). We argue that such advanced mechanisms are complementary to a feature-based solution and

features can be used to improve these mechanisms.

6. SUMMARY

Dynamic software product lines (DSPLs) combine concepts of adaptive systems and software product lines (SPLs) to provide dynamic variability. However, current DSPL approaches commonly use coarse-grained components to implement variability. This reduces customizability and thus limits applicability of a DSPL. We presented an approach that integrates traditional SPLs and DSPLs more closely. Based on a feature-oriented implementation of an SPL and a customizable adaptation framework, we *generate DSPLs* by statically composing features. As in traditional SPLs, we support fine-grained static customization for efficiency reasons; as in DSPLs, we provide adaptability at runtime by *generating* coarse-grained *dynamic binding units*. A dynamic binding unit is tailored to an application scenario by including only user-defined features. For runtime adaptation, we propose to describe adaptation rules in terms of features. We integrate this feature-based adaptation mechanism with our approach for generating DSPLs by transforming the feature model of an SPL according to the binding units of the generated DSPL.

By using features to describe adaptation rules and configuration changes, our approach is independent of an SPL's implementation. Using a feature model, we efficiently validate program adaptations before modifying a configuration at runtime. Our integration of static binding and DSPLs reduces the overhead for dynamic binding. Furthermore, it avoids unneeded dynamic variability, which simplifies runtime adaptation.

In future work, we will integrate our work on optimizing non-functional properties of SPLs [26]. We thus extend the dynamic variant selection process to optimize a variant with respect to non-functional constraints using CSP solvers.

Acknowledgment

We thank Thomas Thüm for comments on drafts of this paper. The work of Marko Rosenmüller and Mario Pukall is funded by German Research Foundation (DFG), project numbers SA 465/34-1 and SA 465/31-2. Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C. Sven Apel's work is supported by the German Research Foundation (DFG), projects AP 206/2-1 and AP 206/4-1.

7. REFERENCES

[1] V. Alves, D. Schneider, M. Becker, N. Bencomo, and P. Grace. Comparitive Study of Variability Management in Software Product Lines and Runtime Adaptable Systems. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9–17, 2009.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009.

[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2005.

[4] D. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer-Verlag, 2005.

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

[6] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. In *International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *Lecture Notes in Computer Science*, pages 491–503. Springer-Verlag, 2005.

[7] N. Bencomo, P. Sawyer, G. S. Blair, and P. Grace. Dynamically Adaptive Systems are Product Lines too: Using Model-Driven Techniques to Capture Dynamic Variability of Adaptive Systems. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–32. IEEE Computer Society, 2008.

[8] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Using Feature Models for Developing Self-Configuring Smart Homes. In *International Conference on Autonomic and Autonomous Systems*, pages 179–188. IEEE Computer Society, 2009.

[9] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged Configuration Using Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283. Springer-Verlag, 2004.

[10] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23:62–70, 2006.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *Computer*, 37(10), 2004.

[13] M. L. Griss. Implementing Product-Line Features with Component Reuse. In *Proceedings of the International Conference on Software Reuse (ICSR)*, volume 1844 of *Lecture Notes in Computer Science*, pages 137–152. Springer-Verlag, 2000.

[14] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic Software Product Lines. *Computer*, 41(4):93–95, 2008.

[15] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using Product Line Techniques to Build Adaptive Systems. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 141–150. IEEE Computer Society, 2006.

[16] H. Härtig, S. Zschaler, M. Pohlack, R. Aigner, S. Göbel, C. Pohl, and S. Röttger. Enforceable Component-based Realtime Contracts. *Real-Time Syst.*, 35(1):1–31, 2007.

[17] J. Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products

in Product Line Engineering. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. IEEE Computer Society, 2006.

[18] I. Mahgoub and M. Ilyas. *Smart Dust: Sensor Network Applications, Architecture, and Design*. CRC Press, 2006.

[19] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. Software Engineering Institute, 2009.

[20] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models at Runtime to Support Dynamic Adaptation. *Computer*, 42:44–51, 2009.

[21] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Wolf. An Architecture-based Approach to Self-adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, 1999.

[22] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[23] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer-Verlag, 1997.

[24] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible Feature Binding in Software Product Lines. *Automated Software Engineering – An International Journal*, 2011. to appear.

[25] M. Rosenmüller, N. Siegmund, T. Thüm, and G. Saake. Multi-Dimensional Variability Modeling. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 11–20. ACM Press, 2011.

[26] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, and G. Saake. Measuring Non-functional Properties in Software Product Lines for Product Derivation. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 187–194. IEEE Computer Society, 2008.

[27] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Society, 2009.

[28] P. Trinidad, A. Ruiz-Cortés, and J. Peña. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *International Workshop on Dynamic Software Product Line*, pages 51–56. Kindai Kagaku Sha Co. Ltd., 2007.