Tailoring Dynamic Software Product Lines

Marko Rosenmüller, Norbert Siegmund, Mario Pukall
University of Magdeburg, Germany

Sven Apel
University of Passau, Germany

Abstract

Software product lines (SPLs) and *adaptive systems* 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 bound statically before runtime. By contrast, adaptive systems support feature binding at runtime and are sometimes called *dynamic SPLs (DSPLs)*. DSPLs are usually built from coarse-grained components, which reduces the number of possible application scenarios. To overcome this limitation, we closely integrate static binding of traditional SPLs and runtime adaptation of DSPLs. We achieve this integration by statically generating a tailor-made DSPL from a highly customizable SPL. The generated DSPL provides only the runtime variability required by a particular application scenario and the execution environment. The DSPL supports self-configuration based on coarse-grained modules. We provide a feature-based adaptation mechanism that reduces the effort of computing an optimal configuration at runtime. In a case study, we demonstrate the practicability of our approach and show that a seamless integration of static binding and runtime adaptation reduces the complexity of the adaptation process.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries, Reuse models

General Terms Design, Languages

Keywords Software Product Lines, Dynamic Binding, Feature-oriented Programming

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) [28]. SPL engineers consider *features* as central abstractions for configuration because they are implementation independent and map directly to user requirements. For example, a feature QUERYENGINE of an SPL for database management systems (DBMS) represents functionality to execute queries using the structured query language (SQL). In SPL engineering, features are usually *bound* statically. That is, a user selects the desired features and a generator creates the corresponding software product containing exactly the needed features.

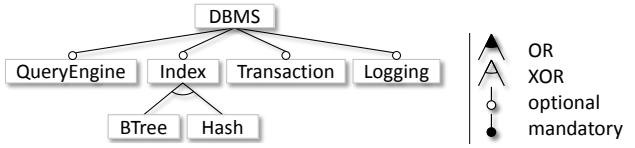

Figure 1. Feature model of a simple DBMS.

Valid feature combinations are often described in a *feature model* using a hierarchical representation of an SPL's features and constraints between them [12]. In Figure 1, we depict an example of the feature model of a *database management system (DBMS)*. Mandatory and optional features are denoted by filled and empty bullets. To specify invalid feature combinations, domain engineers define relations between features, such as OR and XOR, plus 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.

In contrast to an SPL, an adaptive system offers variability at runtime in order to adapt to changing requirements [18]. Approaches for runtime adaptation are often based on components and describe program adaptations at the architectural level [27]. They allow a programmer to specify adaptation rules for reconfiguring components and thereby abstract from the concrete implementation [14, 16, 20]. *Dynamic SPLs (DSPLs)* integrate concepts of SPLs and adaptive systems [1, 8]: The products of a DSPL can be reconfigured at runtime. In contrast to describing program adaptations using architectural models, there are DSPL approaches that support feature-based runtime adaptation. For example, some approaches use *feature models* to describe dependencies between features and to reason about runtime variability of DSPLs [10, 19, 21, 37]. Describing also program adaptations in terms of features abstracts from implementation details, simplifies reconfiguration of running programs, and allows for checking consistency of adaptations [10]. Such feature-based approaches use a mapping of DSPL features to the components that are used for implementation [21, 37]. However, components are usually coarse-grained and limit customizability of a DSPL. For example, it is imperative to customize components for embedded systems to remove unneeded functionality and to tailor the components with respect to the hardware [35]. Increasing customizability with small components is usually not an option due to an increasing communication overhead between the components [9].

We bridge the gap between feature-based variability modeling and component-based runtime adaptation, by integrating generative SPL engineering and DSPLs. In previous work, we have shown how to integrate static and dynamic feature binding by statically merging a set of features into a *dynamic binding unit* according

GPCE'11, October 22–23, 2011, Portland, Oregon, USA.

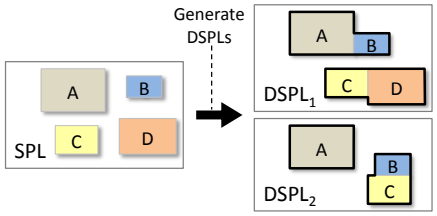

Figure 2. Generating tailor-made DSPLs from the features of an SPL (A–D): For $DSPL_1$ we generate two binding units (A, B) and (C, D); for $DSPL_2$ we generate binding units (A) and (B, C).

to the requirements of an application scenario [30]. At runtime, selected binding units are composed to derive a concrete program. In this paper, we extend our previous work to support runtime adaptation and self-configuration on top of binding units:

- We propose to generate tailor-made DSPLs: As illustrated in Figure 2, we implement an SPL with *feature-oriented software development (FOSD)* [2] and statically compose the required features to derive a DSPL. The DSPL supports reconfiguration at runtime based on coarse-grained dynamic binding units.
- We provide a feature-based approach for runtime adaptation and self-configuration.

In contrast to DSPLs that use components for dynamic binding, we generate tailored binding units from the SPL features (cf. Fig. 2). Due to this fine-grained static customization, we improve reuse and minimize resource requirements of DSPLs. We achieve runtime adaptation with a customizable adaptation framework, called *FeatureAce,* which we integrate into a generated DSPL. The framework supports autonomous self-configuration by using features for describing program adaptations and thus guarantees safety for modifications at runtime. By tailoring and minimizing the number of dynamically-bound modules, we reduce the effort for computing a configuration of a DSPL. We demonstrate practicability of our approach with a prototypical implementation of the adaptation framework and a case study. In summary, our contributions are:

- an approach of statically generating tailor-made DSPLs from SPL features, which improves software reuse (Section 3),
- a customizable adaptation framework including customization of adaptation rules and monitoring code (Section 3.1),
- a feature-based approach of adaptation and self-configuration (Section 4.1), which ensures composition safety (Section 4.2) by means of feature models while using coarse-grained binding units for adaptation,
- a case study that demonstrates practicability of our approach and illustrates the optimization capabilities with respect to reconfiguration and resource consumption (Section 5).

2. Feature-oriented Programming

Using components to implement variability is sometimes too restrictive because components are coarse-grained and thus limit customizability of an SPL. For example, many small features and cross-cutting functionality are hard to implement in individual components [17]. In contrast, *feature-oriented programming (FOP)* [6, 29] can be used to implement the features of an SPL in a modular way. With FOP, one can achieve 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 system. In FOP, features are implemented in *feature modules* as increments in functionality using a one-to-one mapping [6]. A user creates a

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 the features QUERYENGINE and TRANSACTION.

program (a *variant* of an SPL) by selecting features that satisfy requirements. Based on the feature selection (a.k.a., the *configuration*), a generator composes the corresponding feature modules to yield a concrete program.

FeatureC++. FeatureC++[1] is a language extension of C++ that supports FOP [4]. In Figure 3, we depict the FeatureC++ code of a class `DB` of a DBMS (cf. Fig. 1). A programmer typically decomposes a class into smaller class fragments called *base class* and *class refinements* according to the features of the SPL. A base class, such as class `DB` (Lines 1-3), implements basic functionality. For example, method `Put` (Line 2) stores data provided as key-value pairs. A class *refinement* is denoted by keyword `refines` and extends a base class to provide code required for a particular feature. The refinement in feature QUERYENGINE (Lines 4–9) introduces a new member `queryProc` and a new method `ProcessQuery` for processing SQL queries. Feature TRANSACTION overrides method `Put` (Line 12) and invokes the refined method using keyword `super` (Line 14). Based on the implementation shown in Figure 3, we can generate four different DBMS variants by composing different sets of feature modules: We can generate a simple DBMS consisting of CORE only, but we can also derive variants with any combination of the features QUERYENGINE and TRANSACTION.

Static and Dynamic Feature Binding. FeatureC++ supports static binding of features at compile-time and dynamic binding at load-time and runtime [30]. When binding all features of an SPL statically, a single binary is generated for the variant. 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, composing the CORE implementation and feature TRANSACTION of Figure 3 means to generate a single class `DB` that includes all code of the corresponding base class and its refinement in feature TRANSACTION.

For dynamic binding, FeatureC++ generates compound features, called *dynamic binding units*. A dynamic binding unit is tailored to an application scenario and consists of a set of statically merged features. It is similar to a component but it includes only required functionality. To yield a concrete program, a dynamic binding unit is bound as a whole with other dynamic binding units at runtime. At the class level, FeatureC++ supports dy-

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

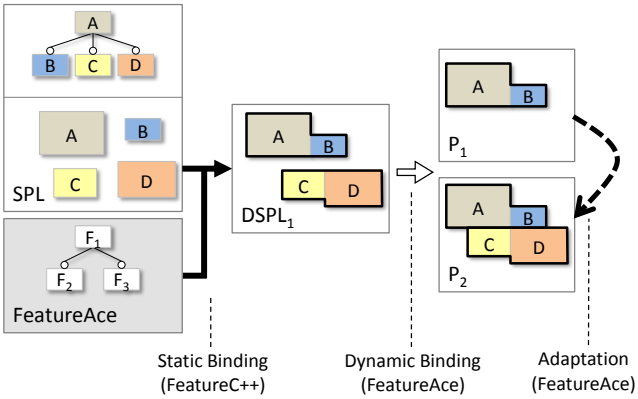

Figure 4. Static composition of an SPL and FeatureAce (➡) resulting in $DSPL_1$ and subsequent dynamic composition of binding units (⇨) resulting in adaptable programs P_1–P_4.

namic binding by generating dynamically composable class fragments. For example, to generate a binding unit that contains the 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 using the *decorator* design pattern [15]. This allows us to change the configuration of a class at runtime, which is the basis for *generating* a DSPL from the features of an SPL. To enable dynamic loading, each binding unit is usually encapsulated in a separate dynamic link library (e.g. a Windows DLL). For a detailed description of dynamic binding units we refer to [30].

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 illustrate in Figure 4 (cf. Fig 2). In $DSPL_1$, two binding units are generated: (A, B) and (C, D). The binding units are composed at runtime to yield the concrete program P_1 or P_2. One of the binding units (e.g., (A, B) in $DSPL_1$) usually acts as the base program, i.e., the part of the DSPL that provides basic functionality that is always needed. Hence, the base program is statically bound and is dynamically extended by additional binding units as required.

The transformation process from an SPL to a running program can be seen as a *staged configuration* [13]: In a first step, FeatureC++ statically merges a set of features into dynamic binding units (➡ in Figure 4). In a second step, the generated binding units are composed at runtime according to a dynamic feature selection (⇨ in Figure 4). Hence, a DSPL comprises a subset of the products of the SPL it was generated from. That is, a DSPL is a *specialization* of a corresponding SPL and provides dynamic variability only.

In contrast to our previous work [30], we support runtime adaptation of programs (i.e., reconfiguration) using a customizable framework, called FeatureAce. FeatureAce is included into a generated DSPL and is responsible for composing features and modifying a program's configuration at runtime. Reconfiguration of a program means to add and remove binding units dynamically. For example, program P_1 of Figure 4 can be reconfigured into P_2 by adding binding unit (C, D). FeatureAce computes the needed configuration changes using a SAT solver. Since FeatureAce itself is developed as an SPL, programmers can choose the required composition and adaptation mechanisms. In the following, we describe FeatureAce and the runtime adaptation process in detail.

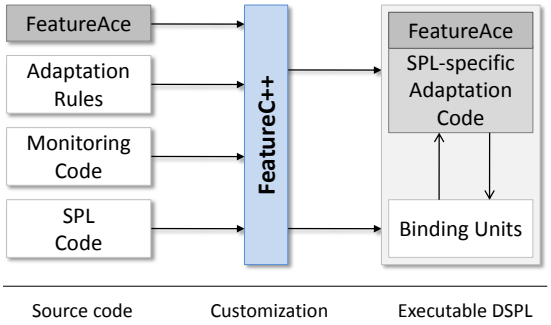

Figure 5. Generating a DSPL from FeatureAce, adaptation rules, monitoring code, and an SPL's implementation.

3.1 FeatureAce: A Customizable Adaptation Framework

FeatureAce is an SPL-independent framework for (re)configuration of DSPLs at runtime. Based on a user-defined feature selection, the FeatureC++ compiler generates a tailor-made DSPL from SPL features and the features of FeatureAce (➡ in Figure 4). FeatureAce is statically bound using generated, SPL-specific glue code. In the executable DSPL, a generic metaprogram (part of FeatureAce) is responsible for controlling the dynamic composition of binding units (⇨ in Figure 4) and for self-adaptation at runtime (e.g., reconfiguring P_1 into P_2 in Figure 4). Programmers describe runtime adaptations with declarative adaptation rules. The execution of a rule is triggered by events spawned in monitoring code of the DSPL. To ensure correctness of program adaptations with respect to the feature model, FeatureAce computes a valid configuration by applying the adaptation rules to the feature model of the DSPL.

In Figure 5, we provide a more detailed view of the transformation process. A user selects the required features from FeatureAce, adaptation rules, monitoring code, and the SPL and the FeatureC++ compiler generates a customized DSPL:

- Users can customize FeatureAce to choose between manual and autonomous adaptation and to enable validation of adaptations if required.
- Adaptation rules are stored in separate feature modules to allow the programmer to choose actually required rules at deployment time, e.g., to choose between alternative adaptation rules.
- Monitoring code of the DSPL that triggers adaptation events is implemented in distinct feature modules. Hence, it is possible to use only required monitoring code and to choose between alternative implementations.

The customization of the adaptation infrastructure allows us to cope with changing requirements (e.g., to support different execution environments). In the following, we use static binding for customization of the adaptation infrastructure. In general, parts of this variability may also be needed at runtime, which requires dynamic binding of selected features of FeatureAce and adaptation code. This is beyond the scope of this paper but it illustrates that there are further challenges for improving the flexibility of our approach.

In Figure 6, we depict 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 DSPL instance (i.e., the configured and running program) and feature SELF-CONFIG supports rule-based self-configuration. Feature VALIDATION checks the validity of an SPL variant before composing the binding units. For customization, a user defines the required adaptation facilities

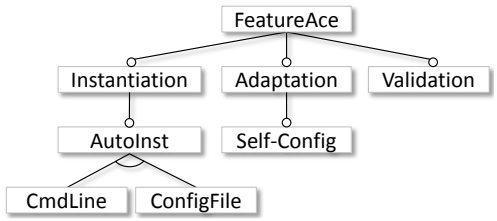

Figure 6. The feature model of FeatureAce. Customization of dynamic product instantiation and runtime adaptation capabilities is achieved by selecting the corresponding features.

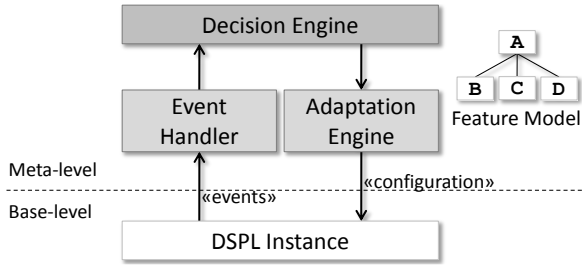

Figure 7. Architecture of a DSPL: Domain code is at the base-level and adaptation code is at the meta-level.

of FeatureAce and may even add user-defined extensions, such as special adaptation mechanisms. FeatureAce extensions are implemented as additional feature modules without the need for invasive modifications of the framework.

As shown in Figure 7, the adaptation metaprogram of FeatureAce provides a decision engine that uses the feature model of the DSPL to ensure validity of changes in the running program. Monitoring code for analyzing the context at runtime is located at the base-level. It is implemented in feature modules of the SPL because it is usually domain-specific. For example, code for monitoring DBMS queries may trigger an event for loading a feature that implements a special search index when a particular kind of query is detected. The events triggered by the monitoring code are captured by an event handler that activates the decision engine. Based on adaptation rules and the feature model, the decision engine computes a new configuration. The adaptation engine applies required configuration changes by loading and unloading binding units.

After generating a DSPL, there is an n-to-1 mapping of the original features to the dynamic binding units of the DSPL. For example, features A and B in Figure 4 map to binding unit (A,B) of $DSPL_1$. In the following, we call the binding units *features of the DSPL* and use a feature model to describe dynamic variability. The DSPL features are used for composition and adaptation at runtime, as we describe next.

3.2 Instantiation and Adaptation of DSPLs

FeatureAce supports a set of operations for instantiation and adaptation of a program from selected DSPL features at runtime:

Instantiation: A program is composed from multiple DSPL feature, implemented as binding units. The result is a stack of feature instances that represents a DSPL instance.

Adaptation: An already running DSPL instance can be modified by adding and removing features as well as activating and deactivating already loaded features.

Note that not every feature that can be bound at load-time can also be bound at runtime without further modification. For example, runtime adaptation requires to support consistent changes with respect to the state of objects. There are further issues, such as concurrency control and state transfer, that have to be considered for transition from an SPL to a DSPL [32]. As a solution, developers may provide special code for binding at runtime, such as *state transformer functions* [26]. Using FOP, this code can be separated in feature modules that are only included in a program when runtime adaptation is used. Implementation of such features is beyond the scope of this paper.

```
1 void ConnectSQLite(string db) {
2   FeatureConfig::ActivateFeature("SQLite"); //activate feature SQLite
3   ConnectDB(db); //continue with activated feature
4 }
```

Figure 8. FeatureC++ source code for activating feature SQLITE from the base-level.

After composing a concrete DSPL instance, FeatureAce provides different adaptation mechanisms:

Add and remove features: A feature can be added to or removed from a running DSPL instance. A feature that is not part of any running DSPL instance can be deleted and unloaded.

Activate and deactivate features: A feature can be deactivated if it is temporarily not needed 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 and can be accessed via an API from an external program or from the base-level of the DSPL itself.

Reflection vs. Rule-based Adaptation. There are two ways to use the adaptation mechanisms of FeatureAce. First, we support manual adaptation by external programs via the API of FeatureAce or by the DSPL itself using *reflection*. Second, we provide a rule-based adaptation mechanism. For manual adaptation, FeatureAce uses the feature model of the DSPL to validate a feature selection at runtime with a SAT solver.[2] When using the rule-based mechanism, we use a SAT solver to derive a valid configuration dynamically. To provide only the actually required variability at runtime, we transform the original SPL feature model according to the generated DSPL, as we describe in Section 4.

Via reflection, the base-level of a DSPL can access the adaptation meta-level for observing or modifying the current configuration. The reflection mechanism is sufficient for simple adaptations, e.g., when events can be directly mapped to a configuration change. In Figure 8, we depict an example of activating a feature for database access in a client by accessing the meta-level from the base-level (Line 2). This mechanism simplifies SPL development since no additional code for an adaptation metaprogram is needed. To separate domain implementation from adaptation mechanisms, programmers should implement such adaptation code in distinct feature modules.

4. Rule-based Program Adaptation

A more flexible mechanism that is independent of the implementation of an SPL is to describe adaptations in a declarative way using adaptation rules. An adaptation rule defines how a configuration of a DSPL must be changed when an event occurs. In contrast to many existing approaches, we use features to define adaptation rules. In particular, an adaptation rule describes constraints (e.g., required features) that the configuration of a program must satisfy after adaptation.

[2] Even though the SAT problem is NP-complete, it has been shown that validity of feature models can be efficiently checked [24].

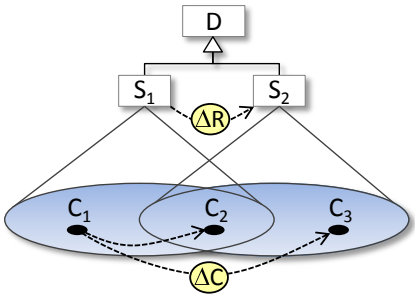

Figure 9. DSPL D with specializations S_1, S_2, configurations C_1–C_3, and adaptation from S_1 to S_2.

4.1 Feature-based Adaptation Rules

A configuration C of a program P of a DSPL is the set of features that is included in P. During adaptation, we derive a configuration C from a set of requirements R that define which features of the DSPL must be included in a valid program. In the simplest case, the requirements are defined by 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 [31]. A configuration constraint is not different from a domain constraint of a feature model but it is added to and removed from the model at runtime. For example, to express that a feature must be included in a program, we can define 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)[3]:

$$R \quad = \quad \{\text{QUERYENGINE}, \text{INDEX}\} \tag{1}$$

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

$$C \quad = \quad \{\text{QUERYENGINE}, \text{INDEX}, \text{HASH}\}. \tag{2}$$

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 program, R defines a *specialization* S of our DSPL that represents multiple configurations, as illustrated in Figure 9. In this example, DSPL D has two specializations S_1 and S_2, which we denote with an empty arrowhead [31]. Each specialization represents multiple configurations (illustrated with a cone). For example, C_1 and C_2 are configurations of S_1. Each specialization represents a subset of the configurations of the unspecialized DSPL.

We can represent a set of requirements R as a single propositional formula using a conjunction of all requirements. For instance, R from equation (1) corresponds to the boolean constraint QUERYENGINE $\wedge$ INDEX. Since a feature model can also be translated into a propositional formula [5], we can check if 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 test if R is a valid set of requirements with respect to FM. This can be done by checking whether we can derive at least a single valid configuration, i.e., $FM \wedge R$ must be satisfiable [36].

Adaptation Rules. The current configuration C of a running program of a DSPL is modified by a configuration change ΔC (i.e., a reconfiguration; cf. Fig. 9) that 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 rule A 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):

$$R' \quad := \quad \Delta R \bullet R \tag{3}$$

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

The modified requirements R' must be satisfied after applying rule A to a configuration C. That is, ΔR does *not* directly modify the configuration of a DSPL but it modifies a set of requirements R that correspond to a specialization of the DSPL. As illustrated in Figure 9, applying ΔR to S_1 results in 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 the configuration with the smallest number of features. Finally, we derive the corresponding configuration change ΔC, which is a pair $(\Delta C_\oplus, \Delta C_\ominus)$. It defines the set of features that must be added $(\Delta C_\oplus)$ and removed $(\Delta C_\ominus)$ for adaptation. We compute it from the current configuration C and the target configuration C':

$$\Delta C \quad := \quad (\Delta C_\oplus, \Delta C_\ominus) \tag{5}$$

$$\Delta C_\oplus \quad := \quad C' \setminus C \tag{6}$$

$$\Delta C_\ominus \quad := \quad C \setminus C' \tag{7}$$

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 used:

$$A \quad = \quad (E_{Range}, (\{\text{BTREE}\}, \emptyset)) \tag{8}$$

$$R' \quad = \quad (\{\text{BTREE}\}, \emptyset)) \bullet R \tag{9}$$

$$= \quad \{\text{QUERYENGINE}, \text{INDEX}, \text{BTREE}\} \tag{10}$$

$$C' \quad = \quad \{\text{QUERYENGINE}, \text{INDEX}, \text{BTREE}\} \tag{11}$$

$$\Delta C \quad = \quad (\{\text{BTREE}\}, \{\text{HASH}\}). \tag{12}$$

Rule A adds feature BTREE to R (i.e., BTREE must occur in a valid configuration), which results in the modified requirement R'. From R' we derive a new configuration C'. The required configuration change ΔC (adding feature BTREE and removing feature HASH) is derived from C and C' according to equations (5–7).

In contrast to modifying the set of requirements, a direct configuration change is too restrictive. For example, an adaptation rule that adds feature BTREE directly to configuration C violates the XOR constraint of the feature model (either BTREE or HASH must be selected). As another example, consider a rule that removes feature BTREE from a configuration because it is not required any longer. Such a rule causes a conflict if the feature to be removed is required by another constraint. With named requirements, we can simply remove the requirement that is not needed and the feature will only be removed from the configuration if not needed due to another requirement.

Specifying Adaptation Rules. We specify adaptation rules in a declarative language, as shown in the example in Figure 10; we depict the corresponding grammar in Figure 11. A rule consists of a name, a named adaptation event E (e.g., `OnTxn` in Line 2) that triggers adaptation, and actions ΔR, which describe the required configuration changes. Currently, we create adaptation events in mon-

[3] Users can provide an initial set of requirements in a configuration file using the declarative language VELVET [31]. The initial set can be empty.

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

Figure 10. Two adaptation rules that add the 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 | "(" Constraint ")" |
10   "!" Constraint | Constraint ConstrOp Constraint ;
11 ConstrOp: "&&" | "||" | "->" | "<->" ;
12 FeatureName: ID ;
```

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

itoring code using the host language. For example, event `OnTxn` in Figure 10 is signaled when monitoring code observes a transaction query. The event triggers loading the transaction management feature as defined in rule `BeginTxn`. An action adds or removes named configuration constraints using keywords `addReq` and `removeReq` 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.

Applying Adaptations. Before computing a new configuration when applying an adaptation rule, FeatureAce checks 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} overlap then there are configurations that can be derived from both specializations (e.g., C_2 in Figure 9). Hence, if the current configuration of the DSPL 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 already 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 at least one valid configuration that satisfies R_{i+1}, the decision engine checks satisfiability of the feature model including the new requirements. If there are multiple valid configurations, the decision algorithm has to choose the *best* one. Which configuration is the *best* depends on the domain, the application scenario, and the context at runtime [14, 38]. 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 [34], such as memory consumption, performance, or quality of service. We currently choose the configuration with the smallest number of configuration changes. For that reason, FeatureAce tries to keep already configured features to minimize changes. Features are removed when they violate a constraint. Hence, when an adaptation rule removes a constraint from requirements R, this does not always cause a reconfiguration. Furthermore, we remove features that are not required for a pre-defined time span to provide a simple mechanism for reducing resource consumption when features are not used anymore.

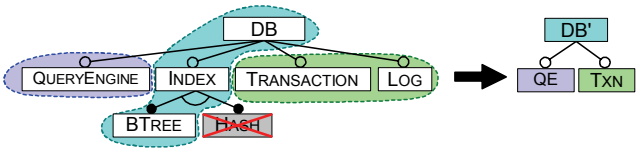

Figure 12. Transformation of a feature model and the corresponding transformation of actions of adaptation rules (lower part) according to defined binding units. Transformed features in adaptation rules are underlined.

To reduce resource consumption, a configuration can also 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 we may optimize a configuration using CSP[4] solvers [7, 38].

4.2 Safety of Runtime Adaptation

Since we merge multiple features of an SPL into a single binding unit of a generated DSPL, there is an n-to-1 mapping of SPL features to DSPL features. For safe adaptation at runtime, however, we have to reason about the dynamic variability provided by the generated DSPL, which represents a subset of the products that can be derived from the SPL. We thus transform the SPL's feature model according to the generated binding units and derive a special feature model of the DSPL. In the upper part of Figure 12, we show an example of such a feature model transformation. A detailed description of the transformation can be found in [30]. Since domain engineers define adaptation rules in terms of SPL features, we apply a corresponding transformation to the adaptation rules (lower part of Figure 12). This transformation is achieved by replacing each feature in the adaptation rules with the binding unit of the feature in the generated DSPL. For example, we have to replace all occurrences of feature TRANSACTION with its corresponding binding unit TXN (first action in Figure 12). After transforming all actions of Figure 12, requirement `RQ` (second action) is always satisfied because `DB'` is the root of the feature tree and included in every configuration. Hence, we can remove all actions that add or remove requirement `RQ`, such as the last two actions in Figure 12.

The correctness of an adaptation rule with respect to the SPL's feature model can be checked with a SAT solver already before runtime. Furthermore, we can check whether there are combinations of events that may result in configuration conflicts due to applying rules at the same time. For example, if two rules add conflicting requirements, only one adaptation would be possible. Unfortunately, this can only be checked for a small number of adaptation rules because there is an exponential number of combinations of adaptation rules and even the order of the adaptations matters. However, due to customization of the DSPL with a reduced number of features and adaptation rules, this may often be possible in practice.

After transforming the feature model including adaptation rules, we can validate the transformation by checking whether all transformed rules can be applied to the DSPL's feature model. For ex-

[4] Constraint satisfaction problem

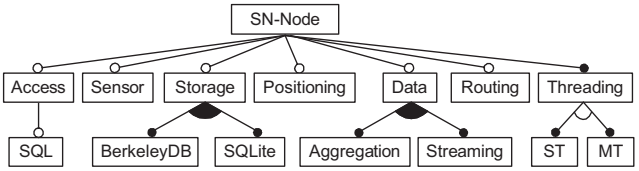

Figure 13. Feature diagram of an SPL for sensor network nodes.

ample, an adaptation rule that adds requirement ¬`Btree` is transformed into a rule with requirement ¬`DB'`, which is not satisfiable since *DB'* is the root of the feature model and is always true. Such a rule is invalid with respect to the generated DSPL (i.e., with respect to the chosen binding units). As another example, consider the features TRANSACTION and LOG in Figure 12. Both features are part of binding unit TXN. Hence, both features are always present at the same time in the generated DSPL. Consequently, an adaptation rule that *requires* LOG and *excludes* TRANSACTION is invalid in this DSPL. Using a SAT solver, we can detect such invalid transformations before runtime.

Finally, the model transformation and the model-based runtime adaptation process guarantee that a new configuration is correct with respect to the original feature model. In combination with static type checking of the entire SPL [3], we can even ensure static type safety for runtime adaptations.

5. Case Study

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

5.1 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.) [23]. There are different types of nodes in a sensor network. *Sensor nodes* measure data, store it locally, and send it to other nodes. *Aggregation nodes* aggregate data (e.g., computing the mean value) from other nodes. *Access nodes* provide access to clients that connect to the network.

With SPL technology, we can *generate* different program variants tailored to the different kinds of SN nodes. In Figure 13, we depict an excerpt of the feature diagram of an SPL for sensor nodes that we implemented in FeatureC++. Subfeatures of DATA are used for aggregation in aggregation nodes and streaming in access nodes. A node does not always play a single role (e.g., being a sensor node) but possibly 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 all physical nodes can play every role. For example, only a node with sufficient storage capacity can be used for data aggregation. Such limitations influence the configuration process when defining binding units at deployment time:

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

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 and their binding units for two kinds of devices.

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.

For evaluation, we use a sensor network simulation, in which each node of the network is simulated by a separate process that communicates with other nodes. The nodes software can be deployed on embedded devices, but for simulation we use a desktop environment.

5.2 Defining Binding Units

In Table 1, we show a sample assignment of roles for two types of node hardware and the corresponding binding units. The binding units are composed from the features of Figure 13. 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).

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 *data aggregator*, or an *access node* 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 a 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 used hardware, application scenarios, etc.

The feature selection influences the binary size of the generated node software. A variant that uses only static binding (binding unit STATICSENSE in Table 1) 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 and a size of 104 KB. This overhead mostly comes from code of the infrastructure for runtime adaptation (i.e., FeatureAce including a SAT-solver), which is independent of the number of features. As we observed in previous studies, the resource consumption increases with an increasing number of binding units [30]; it can be optimized by generating a DSPL that provides only required dynamic variability. 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 apply self-configuration also on resource-constrained devices. Nevertheless, on systems with highly limited resources only static binding is an option. With our approach, a user can choose at deployment time whether to use static binding or to support runtime adaptation.

5.3 Self-Adaptation

Adaptation Rules. We define adaptation rules within dedicated feature modules. For example, we place rules for activating and deactivating stream processing in feature STREAMING. The rules are thus 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 an adaptation 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 from the requirements after all streaming queries have been processed. We do not directly remove the feature because it 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 other 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 a node has been determined.

Reconfiguration. In Figure 14, we depict evaluation results for the adaptation process.[5] We analyzed the time needed for computing whether an adaptation (using a SAT solver) is needed and the time for reconfiguration. To show the benefits of statically optimizing the feature model, we compared reconfiguration of the same 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.

Stream processing is triggered by incoming streaming queries (denoted with b in Fig. 14), which results in a runtime adaptation to load binding unit STREAMING. In our example, 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. Computing 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 0.32 ms when using the complete feature model with 55 features and 0.05 ms for the DSPL model with 6 features (with a 95 % confidence interval of ± 0.13 and ± 0.10 ms respectively). This computation time is required for checking whether the node has to be reconfigured due to the context change. Compared to a reconfiguration that takes 20-50 ms, 0.32 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 (i.e., 85 % less time) for checking whether an adaptation is needed. This demonstrates the importance of reducing the variability for runtime adaptation by optimizing the feature model.

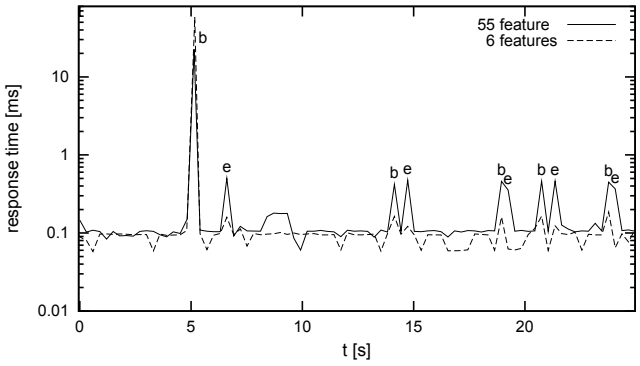

Figure 14. Response time (logarithmic scale) during reconfiguration of a query processing sensor network node using a feature model with 55 features and a simplified model with 6 features. Begin and end of stream processing are denoted with (b) and (e).

5.4 Discussion

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 further optimize the runtime adaptation process, as we discuss next.

Implementation-independent Adaptations. Using features to describe adaptations, we provide an adaptation mechanism that abstracts from the modules actually used for dynamic binding. Hence, we can generate binding units that fit to an application scenario and the execution environment, while being able to reuse adaptation rules. As in component-based approaches, there is an m-to-n mapping of SPL features to dynamic binding units. At compile-time, we simplify this complex mapping by transforming the feature model and adaptation rules according to the generated binding units. The result is a simpler 1-to-1 or 1-to-n mapping of DSPL features to binary modules that must be considered during adaptation. We cannot always achieve a 1-to-1 mapping, because a compound feature may overlap with other compound features, which results in multiple code units per binding unit. These crosscutting modules correspond to *derivatives* known from feature-oriented software development [22].

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

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

before runtime. Currently, we do not check whether a dynamically bound feature supports binding to an already running SPL instance (e.g., supporting state transfer; cf. Sec 3.2). This could be achieved with an extension of the feature model that allows for annotating such features.

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 analyzed in previous work [30].

Computational Complexity. We have shown that we can reduce computations for reconfiguration at runtime in two ways: (1) by avoiding unneeded adaptations and (2) by simplifying 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 an adaptation is needed can easily require more computing power than available. Hence, it is important to simplify the feature model.

Including also non-functional requirements in these computations is a challenging task with respect to computational complexity [14]. Since the computation time to solve constraint satisfaction problems increases exponentially with the number of features [7, 38], it is even more important to reduce the number of binding units as far as possible. Our approach reduces the overall complexity and can be combined with CSP solvers to consider also numerical constraints (e.g., memory restrictions) when computing an optimal configuration at runtime [33]. Further simplification is possible by caching the results of a SAT or CSP solver.

6. Related Work

There are several approaches that use components and architecture-based runtime adaptation as proposed by Oreizy et al. [27]. We abstract from implementation details by using features for configuring a program at runtime. This allows us to reason about configuration changes at runtime at a conceptual level and to describe adaptation rules in a declarative way without taking the high-level architecture into account. There are also approaches that apply SPL concepts to develop adaptive systems, e.g., using feature-oriented concepts for modeling dynamic variability [10, 14, 19, 21, 37]. We aim at building a foundation for integrating existing approaches using features for SPL configuration *and* runtime adaptation. In the following, we compare our approach with respect to the most prominent related approaches.

Some approaches describe dynamic variability in terms of features [10, 21, 37]. Lee et al. use a feature model to describe static and dynamic variability of an SPL. They suggest to manually develop components (i.e., feature binding units) for implementing dynamic variability [21]. We decide *at build-time* which binding time to use and use features (i.e., not implementation units) for specifying adaptations and for validating a configuration. Furthermore, a component-based approach requires mapping features to components. We resolve the mapping before deployment by transforming the feature model and the adaptation rules accordingly.

Floch and Hallsteinsen et al. present with MADAM an approach for runtime adaptation that uses SPL techniques as well as architectural models [14, 19]. They propose to model variability using component-based SPL techniques [19]. We use features for modeling variability and runtime adaptation to further abstract from the underlying implementation and architecture.

Cetina et al. use feature models to describe the variability of an adaptive system [10]. To adapt a system, they modify a configuration by adding or removing features. This results in the problems discussed in Section 5.4, 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. The result is a transformed feature model for runtime adaptation that represents the required variability only.

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) [25]. They use aspects to describe model adaptations and reconfigure the underlying program based on changes of the model. By 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.

Safe composition is important for component-based software development. For example, the Treaty framework combines verification of component assemblies using contracts and event-condition-action rules [11]. By contrast, we use feature models for validating configurations at runtime. Due to the use of FOP, we can furthermore check type-safety for an entire SPL before deployment [3]. In contrast to component-based approaches, we do not provide any means for verifying compositions based on contracts, but we think that approaches for verification are orthogonal to a feature-oriented approach for composition and can be integrated.

We use FOP for implementing adaptive systems, but our approach for feature-based adaptation may also be used with other implementation units such as aspects [10, 21, 25, 37]. In this case, we can still use features to describe adaptation changes. After a new configuration has been derived and validated, a corresponding set of components has to be determined. Some component approaches provide advanced capabilities for runtime adaptation not considered here (e.g., adaptation planning, state transfer, etc.). Such advanced mechanisms are complementary to a feature-based solution and features can be used to improve these mechanisms by abstracting from implementation details.

7. Summary

Current approaches for developing dynamic software product lines (DSPLs) commonly use coarse-grained components to implement variability. This reduces customizability and thus limits applicability of a DSPL. We presented an approach that allows us to tailor DSPLs by closely integrating traditional, static SPLs and DSPLs. Based on a feature-oriented implementation of an SPL and a customizable adaptation framework, we *generate tailor-made DSPLs* by using fine-grained features to statically generate coarse-grained *dynamic binding units*. As in traditional SPLs, we support fine-grained static customization for efficiency reasons; as in DSPLs, we provide adaptability at runtime by composing dynamic binding units at runtime. A dynamic binding unit is tailored to an application scenario by including only user-selected features.

For runtime adaptation, we describe adaptation rules in terms of features. We provide a feature-based adaptation mechanism by transforming the feature model of an SPL according to the binding units of the generated DSPL. By using a feature model to derive a valid configuration at runtime, our approach is independent of an SPL's implementation. Our integration of static binding and DSPLs reduces the overhead for dynamic binding, avoids unneeded dynamic variability, and simplifies computations at runtime.

In future work, we will integrate our work on optimizing non-functional properties of SPLs [33, 34]. This means to extend the dynamic variant selection process to optimize a variant with respect to non-functional constraints using CSP solvers.

Acknowledgments

We thank Thomas Thüm for comments on earlier drafts of this paper. Marko Rosenmüller and Mario Pukall are 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 01IM10002B. Sven Apel's work is supported by the German Research Foundation (DFG), project numbers AP 206/2 and AP 206/4.

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 *Proc. Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9–17. University of Duisburg-Essen, 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, C. Kästner, A. Größlinger, and C. Lengauer. Type Safety for Feature-oriented Product Lines. *Automated Software Engineering*, 17(3):251–300, 2010.

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

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

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

[7] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. In *Proc. Int'l. Conf. Advanced Information Systems Engineering (CAiSE)*, volume 3520 of *LNCS*, pages 491–503. Springer, 2005.

[8] 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 *Proc. Int'l. Software Product Line Conf. (SPLC)*, pages 23–32. IEEE CS, 2008.

[9] T. J. Biggerstaff. A Perspective of Generative Reuse. *Annals of Software Engineering*, 5(1):169–226, 1998.

[10] C. Cetina, P. Giner, J. Fons, and V. Pelechano. Using Feature Models for Developing Self-Configuring Smart Homes. In *Proc. of Int'l. Conf. Autonomic and Autonomous Systems (ICAS)*, pages 179–188. IEEE CS, 2009.

[11] B. D. Claas Wilke, Jens Dietrich. Event-Driven Verification in Dynamic Component Models. In *Proc. Int'l. Workshop on Component-Oriented Programming (WCOP)*, pages 79–86. Karlsruher Institut für Technologie (KIT), 2010.

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

[13] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, volume 3154 of *LNCS*, pages 266–283. Springer, 2004.

[14] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using Architecture Models for Runtime Adaptability. *IEEE Software*, 23(2):62–70, 2006.

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

[16] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-Based Self Adaptation with Reusable Infrastructure. *Computer*, 37(10):46–54, 2004.

[17] M. L. Griss. Implementing Product-Line Features with Component Reuse. In *Proc. Int'l. Conf. Software Reuse (ICSR)*, volume 1844 of *LNCS*, pages 137–152. Springer, 2000.

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

[19] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using Product Line Techniques to Build Adaptive Systems. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, pages 141–150. IEEE CS, 2006.

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

[21] J. Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, pages 131–140. IEEE CS, 2006.

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

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

[24] M. Mendonca, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proc. Int'l. Software Product Line Conf. (SPLC)*, pages 231–240. Software Engineering Institute, 2009.

[25] B. Morin, O. Barais, J.-M. Jezequel, F. Fleurey, and A. Solberg. Models at Runtime to Support Dynamic Adaptation. *Computer*, 42(10):44–51, 2009.

[26] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical Dynamic Software Updating for C. In *Proc. Int'l. Conf. Programming Language Design and Implementation (PLDI)*, pages 72–83. ACM Press, 2006.

[27] 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.

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

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

[30] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible Feature Binding in Software Product Lines. *Automated Software Engineering*, 18(2):163–197, 2011.

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

[32] K. Schmid and H. Eichelberger. From Static to Dynamic Software Product Lines. In *Int'l. Workshop on Dynamic Software Product Lines (DSPL)*, pages 33–38. IEEE CS, 2008.

[33] N. Siegmund, M. Rosenmüller, C. Kästner, P. Giarrusso, S. Apel, and S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines. In *Proc. of Int'l. Software Product Lines Conf. (SPLC)*, 2011. to appear.

[34] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal*, to appear, 2011.

[35] A. Tešanović, K. Sheng, and J. Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *Proc. of Int'l. Database Engineering and Applications Symposium (IDEAS)*, pages 291–301. IEEE CS, 2004.

[36] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proc. Int'l. Conf. Software Engineering (ICSE)*, pages 254–264. IEEE CS, 2009.

[37] P. Trinidad, A. Ruiz-Cortés, and J. Peña. Mapping Feature Models onto Component Models to Build Dynamic Software Product Lines. In *Int'l. Workshop on Dynamic Software Product Lines (DSPL)*, pages 51–56. Kindai Kagaku Sha Co. Ltd., 2007.

[38] J. White, B. Dougherty, and D. C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.