Autom Softw Eng
DOI 10.1007/s10515-011-0080-5

Flexible feature binding in software product lines

Marko Rosenmüller · Norbert Siegmund · Sven Apel · Gunter Saake

Received: 9 July 2010 / Accepted: 18 January 2011

Abstract A *software product line* (*SPL*) is a family of programs that share assets from a common code base. The programs of an SPL can be distinguished in terms of *features*, which represent units of program functionality that satisfy stakeholders' requirements. The features of an SPL can be *bound* either *statically* at program compile time or *dynamically* at run time. Both binding times are used in SPL development and have different advantages. For example, dynamic binding provides high flexibility whereas static binding supports fine-grained customizability without any impact on performance (e.g., for use on embedded systems). However, contemporary techniques for implementing SPLs force a programmer to choose the binding time already when designing an SPL and to mix different implementation techniques when multiple binding times are needed. We present an approach that integrates static and dynamic feature binding seamlessly. It allows a programmer to implement an SPL once and to decide per feature at deployment time whether it should be bound statically or dynamically. Dynamic binding usually introduces an overhead regarding resource consumption and performance. We reduce this overhead by statically merging features that are used together into *dynamic binding units*. A program can be configured at run time by composing binding units on demand. We use feature models to ensure that only valid feature combinations can be selected at compile *and*

M. Rosenmüller (✉) · N. Siegmund · G. Saake
School of Computer Science, University of Magdeburg, Magdeburg, Germany
e-mail: rosenmue@ovgu.de

N. Siegmund
e-mail: nsiegmun@ovgu.de

G. Saake
e-mail: saake@ovgu.de

S. Apel
Department of Informatics and Mathematics, University of Passau, Passau, Germany
e-mail: apel@uni-passau.de

Published online: 03 February 2011

at run time. We provide a compiler and evaluate our approach on the basis of two non-trivial SPLs.

Keywords Software product lines · Feature binding time · Feature-oriented programming · Feature composition · Static binding · Dynamic binding

1 Introduction

Software product line (SPL) engineering has been successfully applied to many domains.[1] An SPL is a family of similar programs that are distinguished in terms of *features*. A *feature* is a unit of program functionality that satisfies a requirement, implements a design decision, and provides a potential configuration option (Apel and Kästner 2009). In *feature-oriented software development (FOSD)* (Apel and Kästner 2009), the programs of an SPL are generated by composing modules that implement features. Depending on the underlying modularization and composition mechanism, features are either *bound statically* (e.g., at compile time or in a preprocessing step) or *dynamically* (e.g., when loading a program or at run time).

Both binding times have benefits: static binding facilitates customizability without any overhead at run time, whereas dynamic binding allows a programmer to flexibly select and bind features even at run time, however, at the cost of performance and memory consumption (Anastasopoules and Gacek 2001). Beside resource consumption and flexibility, there are other reasons that force programmers to use either static or dynamic binding. For example, when it is not known before run time whether a feature is needed or not, programmers can use dynamic binding to avoid deployment of all possible features. On the other hand, not every feature can be bound dynamically. For instance, some features related to the build and execution environment must be chosen before compilation. Examples are the supported CPU architecture and compiler, whether debugging support is needed, or which operating system libraries are used. These decisions have to be made already at build time. We conclude that both binding times are required in SPL development.

With commonly used SPL implementation techniques (e.g., preprocessors-based or component approaches), a programmer is forced to choose between static and dynamic binding already at design time of an SPL (Lee and Kang 2006). The programmer must select an implementation technique that corresponds to the chosen binding time. Changing the binding time after development is not possible. However, relying only on a single binding time restricts the applicability of an SPL. We present an approach that seamlessly integrates static and dynamic binding. We demonstrate how to bind features of an SPL dynamically or statically using the same code base. In contrast to our previous work (Rosenmüller et al. 2008), we can choose a distinct binding time *per feature* after development. We achieve this by statically composing the features that are used in combination into a *dynamic binding unit*, which is bound at run time as a whole. Dynamic binding units are similar to components but are generated at compile time from a user-defined set of features. By statically generating binding

[1] http://www.sei.cmu.edu/productlines/plp_hof.html.

units, we achieve fine-grained customizability while maximizing performance; by dynamically applying the binding units, we achieve a high flexibility at run time. Using code transformations, all features can be implemented with the same technique independent of their binding time, which simplifies SPL development and maintenance. The contributions of this article are:

1. We present code transformations for integrating static and dynamic feature binding in SPLs at modeling and implementation level. Our approach allows developers to flexibly switch the binding time per feature using the same code base. In contrast to existing approaches (Chakravarthy et al. 2008; Czarnecki and Eisenecker 2000; Gilani and Spinczyk 2005; Zdun et al. 2007), we statically generate dynamic binding units to reduce the overhead of dynamic binding.
2. We provide composition safety of dynamic binding using a transformed feature model. To generate such a model, we present transformation rules that correspond to the code transformations that we use for creating dynamic binding units.
3. We demonstrate practicality of our approach with an implementation of the transformation system based on feature-oriented programming.
4. Finally, we evaluate our approach regarding customizability and resource consumption.

2 Feature binding in software product lines

Feature binding is the process of including features in a concrete program at a specific point in time, called the *binding time* (Czarnecki and Eisenecker 2000). There are different possibilities to categorize the binding time of features in SPLs. We refer to *static binding* if a feature is bound in a program before load time (e.g., at compile time) and to *dynamic binding* if it is applied at load time or after loading a program. For example, the C/C++ preprocessor is frequently used to support static binding in SPLs for embedded systems. The preprocessor removes unneeded code from a program before compilation. In contrast, components and plugins support dynamic binding at load time or run time of a program. In the following, we analyze static and dynamic binding to motivate that a combination of both binding times is needed.

Static and dynamic feature binding Most SPL implementation techniques support either static or dynamic binding of features. However, different features may require different binding times (van Gurp et al. 2001) and different application scenarios may require the same feature to be bound at different times. Using static or dynamic binding exclusively is often not feasible for several reasons. For example, static binding cannot be used if required features are not available or not known at deployment time, as it is the case for third-party extensions. Dynamic binding enables independent deployment of features or extensions of a program without rebuilding the whole program. It even provides means for loading extensions on demand (e.g., from a network) or when the configuration of an SPL has to be changed at run time such as in dynamic SPLs (Hallsteinsen et al. 2008).

However, it is also not feasible to rely exclusively on dynamic binding, e.g., for platform- or compiler-specific features. This would limit possible target platforms of an SPL. Furthermore, some devices (e.g., deeply embedded systems) cannot load executable code at run time. Hence, they only support dynamic binding of already loaded code (e.g., using conditional statements), which reduces the benefits of dynamic binding. Finally, dynamic binding usually means a higher development and maintenance effort, which makes it the more expensive alternative.

Resource consumption Both, static and dynamic binding, have benefits and drawbacks with respect to resource consumption (e.g., CPU utilization, memory consumption) of a program. Often only a subset of the features of a program is used at the same time and some features may not be used at all. For example, the required functionality of a *database management system (DBMS)*, deployed on a smartphone, depends on the requirements of the applications that use the DBMS. A Web browser that stores encrypted passwords in a database requires a DBMS with data encryption. If the Web browser is never used, the encryption feature of the DBMS is not required at all. The presence of unused functionality can be avoided by loading features on demand and bind them dynamically. Unfortunately, dynamic binding introduces an overhead to support dynamic composition (Czarnecki and Eisenecker 2000). Hence, depending on the binding time, we observe either a *functional* or a *compositional* overhead:

- *Static binding* causes a *functional overhead* due to features that are not used but present in a program. This results in increased binary size, working memory consumption, and execution time (e.g., due to execution of initialization code).
- *Dynamic binding* introduces a *compositional overhead*, which is caused by glue code, indirections for binding features at run time, and code of the composition infrastructure. This additional code increases binary size and execution time of a program. It may also cause a higher memory consumption at run time (e.g., for storing virtual function pointers in C++).

Mixing static and dynamic binding To cope with the limitations of current implementation techniques, different approaches for static and dynamic binding are combined in practice, e.g., in the Apache Web server,[2] Mozilla (van Gurp et al. 2001), and Oracle's embedded *database management system (DBMS)* Berkeley DB.[3] In these systems, the programmers use the C/C++ preprocessor for static binding (e.g., for platform specific features) and proprietary mechanisms for dynamic binding. For example, the Apache Web server comes with a special module system to load extensions dynamically.

In general, mixing dynamic and static binding provides several benefits. Dynamic binding can be used to achieve extensibility (e.g., for plugins), while other features, such as platform-specific features, are bound statically. Applying static binding to dynamically bound components allows a user to customize the components, e.g., when many components are affected by a *crosscutting feature*. This is problematic when using only dynamic binding (Griss 2000). However, mixing different implementation

[2] http://httpd.apache.org/.

[3] http://www.oracle.com/technology/products/berkeley-db/.

mechanisms increases complexity of SPL development:

- it forces developers to use different implementation mechanisms for different features within an SPL (e.g., preprocessors, design patterns, components, aspect-oriented programming (Kiczales et al. 1997), etc.),
- a developer has to choose the binding time per feature at design time before development (Lee and Kang 2006),
- it hinders reuse because a feature developed for a particular binding time cannot be easily reused in an application scenario or in a different SPL that requires a different binding time.

To summarize, an integration of static and dynamic binding is already used in practice, but without proper support at the implementation level. Relying on a single binding time may increase execution time and memory consumption, which is unacceptable when resources are limited. A combination of both binding times can reduce resource consumption and improve customizability. Additionally, supporting different binding times based on the same implementation mechanism simplifies SPL development and maintenance. Changing the binding time of an implementation unit is supported by a few approaches (Chakravarthy et al. 2008; Czarnecki and Eisenecker 2000; Gilani and Spinczyk 2005; Zdun et al. 2007). However, these approaches do not consider the specifics of SPLs. For example, they do not address static and dynamic binding of entire features. Furthermore, the approaches only support either static or dynamic binding of a code unit, but do not support static composition between multiple dynamically bound units. Moreover, there is no approach that integrates static and dynamic binding with respect to the SPL configuration process. For example, existing approaches do not offer a mechanism to validate a configuration with respect to the feature model statically and dynamically.

3 Integration of static and dynamic binding

We address the trade-off between static and dynamic binding with an approach that seamlessly integrates both binding times. Our approach allows a programmer to choose the binding time *per feature* and to optimize resource consumption and customizability. It is based on *feature-oriented programming (FOP)* (Prehofer 1997; Batory et al. 2004), which allows programmers to modularize the features of an SPL. We implemented our approach on top of *FeatureC++*,[4] an FOP extension of the C++ programming language. We first introduce FOP and describe how we achieve static and dynamic binding of features based on the same feature-oriented implementation of an SPL. We continue with an approach that integrates both binding times at modeling and implementation level.

3.1 Feature-oriented programming

A *feature model* describes the variability of an SPL, as shown in Fig. 1 for a DBMS. It is a hierarchical representation of mandatory (shown with a filled bullet) and optional (shown with an empty bullet) features and relations between them (Czarnecki

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

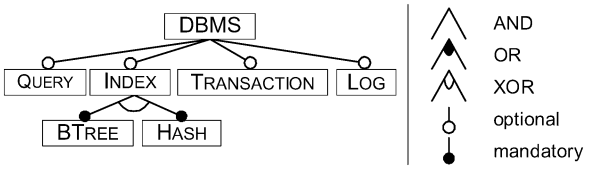

Fig. 1 Feature model of a simple DBMS

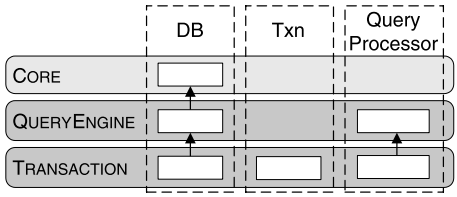

Fig. 2 Decomposition of classes (dashed rectangles) along features (horizontal layers)

and Eisenecker 2000). A feature model thus defines the valid configurations of an SPL. For example, the XOR relation between features BTREE and HASH means that exactly one of the features has to be selected when their parent feature INDEX is selected.

In FOP, features are implemented as increments in functionality (Batory et al. 2004). A user specifies a program by selecting a set of features that satisfy her requirements. Based on the feature selection, a generator composes the corresponding *feature modules* (i.e., the modularized implementation of a feature) to yield a concrete program. In Fig. 2, we depict a decomposition of a DBMS along multiple features (displayed as layers). The DBMS consists of a CORE implementation and two features QUERYENGINE and TRANSACTION. The two features cut across the implementation of the classes `DB`, `Txn`, `QueryProcessor` (shown as dashed boxes). A programmer thus decomposes a class into a *base class* and *class refinements*, shown as white boxes in Fig. 2. Refinements implement extensions of a base class necessary for a particular feature. For example, the base implementation of class `DB` is defined in CORE and extended in QUERYENGINE and TRANSACTION (depicted with arrows).

In Fig. 3, we depict the FeatureC++ code of class `DB` (cf. Fig. 2). Method `Put` is used to store data provided as key-value pairs. The refinement in feature QUERYENGINE (Lines 5–10) adds a new field `queryProc` and a new method `ProcessQuery` for processing SQL queries. Feature TRANSACTION defines a *refinement* of method `Put` (Line 14) and invokes the refined method using the keyword `super` (Line 16). Based on the implementation shown in Fig. 2, we can generate different DBMS variants by composing a varying set of feature modules. For example, we can generate a basic DBMS consisting only of the CORE implementation but we can also derive variants that include QUERYENGINE or TRANSACTION by composing the corresponding feature modules.

With FeatureC++, the feature modules of an SPL can be bound *either* statically at compile time *or* dynamically at load time or run time. To support static binding, the FeatureC++ compiler composes the code of a base classes and selected refinements into a single class. Dynamic binding is implemented by a code transformation that uses the decorator pattern (Gamma et al. 1995) to generate a dynamically composable class fragment for each refinement (Rosenmüller et al. 2008). We describe both

Feature CORE

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

Feature QUERYENGINE

```
5  refines class DB {
6      QueryProcessor queryProc;
7      bool ProcessQuery(String& query) {
8          return queryProc.Execute(String& query);
9      }
10 };
```

Feature TRANSACTION

```
12 refines class DB {
13     Txn* BeginTransaction() { ... }
14     bool Put(Key& key, Value& val) {
15         ... //transaction-specific code
16         return super::Put(key,val);
17     }
18 };
```

Fig. 3 FeatureC++ code of class DB decomposed along the three features CORE, QUERYENGINE, and TRANSACTION

transformations in the following and combine them to integrate static and dynamic binding, as described afterwards.

3.2 Support for static feature binding

To support static feature binding, the classes of an SPL have to be composed at compile time according to the features selected in the configuration process. The FeatureC++ compiler uses a source-to-source transformation from FeatureC++ to plain C++ code. It composes the entire code of the base implementation of a class and its refinements of all selected features into one compound C++ class. This compound class consists of:

- the union of all member variables,
- one method for each method refinement,
- one constructor and destructor for each different constructor/destructor definition, and
- one method for each constructor/destructor refinement.

In Fig. 4, we depict the generated C++ code of a DBMS variant that corresponds to the FeatureC++ code of class DB in Fig. 3. We show the generated code only for illustration and it does not have to be read by a programmer that uses FeatureC++. The code corresponds to a composition of the CORE implementation with feature TRANSACTION. All methods and fields of the corresponding refinements of class DB (i.e., except the code of feature QUERYENGINE) are composed into a single C++ class. The base implementation of method `Put` (feature CORE) was renamed to `Put_Core` (Line 2) to provide a unique name for every C++ method. It is called

```
1  class DB {
2      bool Put_Core(Key& key, Value& val) { ... }
3
4      Txn* BeginTransaction() { ... }
5
6      bool Put(Key& key, Value& val) {
7          ... //Transaction-specific code
8          return Put_Core(key,val);
9      };
10 };
```

Fig. 4 Generated C++ source code of class DB using static binding of CORE and TRANSACTION

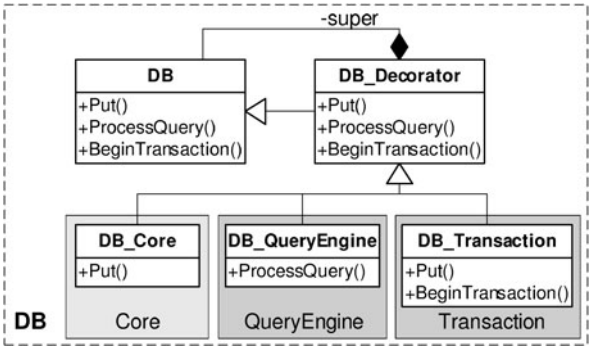

Fig. 5 Generated decorator hierarchy for class DB to support dynamic binding of the features QUERYENGINE and TRANSACTION

from its refinement in Line 8. Using this transformation and by avoiding virtual methods (which are needed in a component-based approach), a C++ compiler can easily inline method refinements. For example, method `Put_Core` is inlined in method `Put` and does not introduce any overhead for method calls. Based on such optimizations, we have shown that FeatureC++ generates code that achieves the same performance as C++ code that does not provide such fine-grained customizability (Rosenmüller et al. 2009).

3.3 Support for dynamic feature binding

For dynamic binding of features, we have to compose the classes of a program at run time according to the dynamic feature selection. We achieve this dynamic composition of a class by transforming each class refinement into a dynamically composable class fragment. For example, when creating a DBMS with transaction management at run time, we have to dynamically compose the base implementation of class DB (cf. Fig. 3) with its refinement in feature TRANSACTION. Similar to the *Delegation Layers* approach (Ostermann 2002), we transform the refinement chain of a class into a delegation hierarchy (Rosenmüller et al. 2008). By using the *decorator pattern* (Gamma et al. 1995) for implementing refinements, we are able compose classes dynamically by composing the generated decorators. Composing two features thus means to compose all classes and class refinements of the features.

For illustration, we depict the class diagram of the transformed class DB in Fig. 5. The base code of class DB and all of its refinements have been transformed into

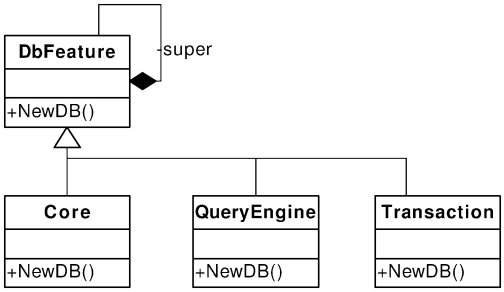

Fig. 6 Generated feature decorators. `DbFeature` is an abstract decorator for the features of the DBMS SPL. The concrete decorators `Core`, `QueryEngine`, and `Transaction` implement feature specific code (e.g., for class instantiation)

decorators (`DB_Core`, `DB_QueryEngine`, `DB_Transaction`). Each decorator belongs to a separate feature. The concrete decorators provide the implementation of methods and method refinements. For example, method `Put` (Line 2 in Fig. 3) and its refinement in feature TRANSACTION (Line 14) are transformed into methods of the concrete decorators `DB_Core` and `DB_Transaction` (cf. Fig. 5). The abstract decorator class `DB_Decorator` maintains a reference to the predecessor refinement (`super` reference) and forwards operations to the next decorator that are not implemented by its concrete decorator. Similarly, a method refinement invokes its refined method also by using the `super` reference of the decorator class. To overcome the *self problem* when calling a method of the compound class (Liebermann 1986) (open recursion), each decorator additionally maintains a *self* pointer that refers to the compound object and not the current concrete decorator. We use the generated decorator interface `DB` to reference dynamically composed classes within the transformed code and also from external source code. All created objects contain an additional proxy to support modifications of the object at run time. The proxy is an empty decorator that only forwards method calls to the decorator that implements the first class refinement.

Feature classes When dynamically creating a product from an SPL (a.k.a. *SPL instance*), we compose multiple features according to a given configuration. Each feature usually contains multiple classes and class refinements that have to be composed consistently at the same time. We support this dynamic composition process by representing features as classes, called *feature classes*. Feature classes are generated in the FeatureC++ code transformation process. Much like ordinary classes and refinements, feature classes are also combined using the decorator pattern to enable class instantiation via delegation, as we describe below. In Fig. 6, we shown an example for the DBMS SPL. For each feature module (CORE, QUERYENGINE, and TRANSACTION), we generate a feature decorator that inherits from an abstract decorator `DbFeature`. The abstract decorator is the base class of all feature classes of the product line. It maintains a `super` reference to the predecessor feature in a composed variant. Instantiation of a feature thus means to create an object of the corresponding feature class. Generating an SPL instance means to combine feature instances using `super` references. Hence, an SPL instance is represented by a stack of feature instances. The features are ordered according to the desired feature composition order.

The dynamic composition process occurs at program startup or at runtime. At program startup, an initial SPL instance is created automatically or manually and then executed (Rosenmüller et al. 2008). For example, a user can provide a list of

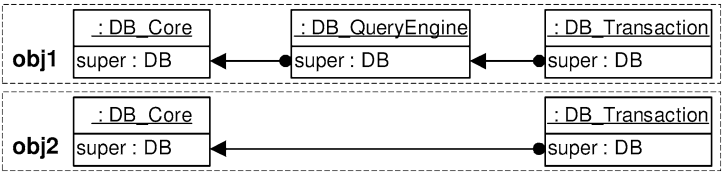

Fig. 7 Object diagrams of the instances `obj1` and `obj2` of two different variants of class `DB` using three and two features respectively

features as program arguments and the corresponding SPL instance is then derived at load time. Manual creation of an SPL instance in user-defined code is done by composing feature instances (i.e., objects of feature classes), as described above.

Class instantiation We use a dynamically composed SPL instance to create objects of the classes of an SPL at run time. Creating an object means to create and combine instances of its class refinements (implemented as decorators; cf. Fig. 5). The required base classes and class refinements are automatically instantiated according to a dynamic feature selection. This instantiation process is realized by generated factory methods of the corresponding feature classes (e.g., `NewDb()` in Fig. 6).

In Fig. 7, we show two examples of objects of class DB that have different sets of refinements. Object `obj1` uses the CORE implementation and refinements defined in features QUERYENGINE and TRANSACTION. Object `obj2` uses only the CORE implementation and the refinement from feature TRANSACTION. Each instantiated refinement contains a `super` reference (cf. Fig. 5) pointing to the next refinement in the chain. For example, the `super` pointer of the instance of `DB_QueryEngine` refers to an instance of `DB_Core`. The dynamically composed objects can be used in the same way as an instance of a regular class. Furthermore, they can be modified at run time by adding or removing decorator instances. Such modifications at run-time are possible because existing object references point to the first decorator in the chain, which is an empty proxy that forwards method calls (omitted in Fig. 7). The refinement chain thus corresponds to a linked list of class fragments. Changing the configuration of the SPL corresponds to insertion, exchange, and deletion of elements of the *refinement list* of the SPL classes.

3.4 Integrating static and dynamic binding

So far, our approach allows programmers only to choose between static and dynamic binding for the entire SPL and not for single features. As discussed in Sect. 2, this approach is still not flexible enough for certain application scenarios. It results in a *functional* or *compositional overhead* depending on the binding time. In previous work, we observed that especially the compositional overhead limits applicability of a pure dynamic approach (Rosenmüller et al. 2008). Next, we extend our approach to integrate both binding times.

3.4.1 Dynamic binding units

We integrate static and dynamic binding by combining multiple dynamically bound features into a single *dynamic binding unit*: Features that are always bound together,

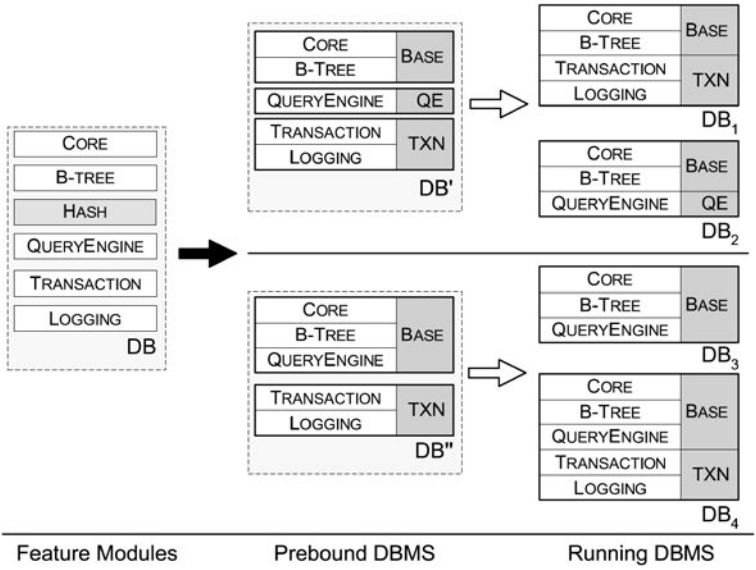

Fig. 8 Examples for static transformations (➡) of a DBMS product line resulting in the prebound SPLs DB' and DB'', and subsequent dynamic composition (⇨) resulting in the running programs DB_1–DB_4. Feature HASH was not selected and is not included in any binding unit

are merged at compile time into a binding unit. This means a two step composition process: First, we use static composition for the features of a binding unit and, second, we compose dynamic binding units in the running program. Static composition results in a *prebound* SPL consisting of a set of dynamically composable binding units, each of which consists of possibly multiple statically composed features. As a binding unit may also contain only a single feature, our approach still supports pure dynamic binding. Rather than manually developing binding units as components (Lee and Kang 2006), we automate this process and generate them on demand at deployment time. That is, a programmer implements an SPL once and chooses the binding time per feature later. With binding units, we reduce the overhead for dynamic binding since the features of a binding unit are statically composed. A binding unit can be bound at any time after program startup.

In Fig. 8, we show an example for generating dynamic binding units. DB' and DB'' denote two prebound SPLs (i.e., not concrete products) after static composition. Feature HASH is not required and is thus not included in any of the prebound SPLs. In DB', feature B-TREE (an index structure for efficient data access) is always required and we thus combine it with feature CORE into a single binding unit BASE. Similarly, TRANSACTION and LOGGING are composed into binding unit TXN. Feature QUERYENGINE is assigned to a distinct binding unit QE. This is different in DB'', which contains the same features bound differently. In DB'', feature QUERYENGINE is not assigned to a distinct binding unit but added to binding unit BASE. From each prebound DBMS, we can create a number of different DBMS (examples DB_1–DB_4) by dynamically composing the binding units according to a given configuration (i.e., a list of desired binding units). Comparing DB_2 and DB_3, we see that both provide

the same functionality but feature QUERYENGINE is contained in a distinct binding unit in DB_2 and is bound statically in DB_3, which leads to differences in flexibility and resource consumption.

Product derivation In summary, the product derivation process of our integrated approach can be divided into three steps: (1) configuration, (2) static transformation, and (3) dynamic composition. In the first step (filled arrow in Fig. 8), a user selects the potentially required features and assigns each feature to a binding unit. In the subsequent static transformation process, the compiler selects the required feature modules and generates dynamic binding units.[5] The FeatureC++ compiler also generates code for composing the binding units at run time. There are two extremes: first, a single binding unit may contain all selected features, which results in a pure statically composed program without any code for dynamic binding. Hence, the product derivation process is finished after static transformation. Second, each binding unit may contain only a single feature resulting in a purely dynamically composable SPL. Between these extremes (which mark the current state of the art), our extended approach supports any combination of static and dynamic binding.

3.4.2 Compound features and feature modeling

When generating dynamic binding units, we generate a prebound SPL with reduced dynamic variability. To ensure consistency of the dynamic composition process, we verify a configuration at run time before composing the binding units. This is done in a similar way as we have shown for pure dynamic composition (Rosenmüller et al. 2008). We use a feature model to avoid invalid configurations. However, because merged features have to be bound as a whole, the final dynamic composition process cannot be based on the SPL's initial feature model. Hence, we have to transform the feature model such that it contains only dynamic variability.

To this end, we represent the static composition process on the modeling level as a transformation of the SPL's feature model. In the following, we first introduce *compound features* to represent binding units at the model level and then describe the required model transformations.

Compound features We represent feature composition by treating features as functions that modify other features or a base program (Lopez-Herrejon et al. 2006; Apel et al. 2010). The composition of one feature with another feature results in a *compound feature*, which is the input for the subsequent composition step. In our case, a dynamic binding unit is a compound feature that is bound at run time. We denote static feature composition with operator $\bullet$ and dynamic feature composition with operator $\circ$. This way, we can describe composition of programs DB_1 and DB_2 (cf. Fig. 8) as follows:

$$Base = BTree \bullet Core \tag{1}$$

[5]Dynamic binding units are stored in the binary of an application or in extension libraries. Currently, we support Windows DLLs.

$$QE = QueryEngine \tag{2}$$

$$TXN = Logging \bullet Transaction \tag{3}$$

$$DB_1 = TXN \circ Base \tag{4}$$

$$= (Logging \bullet Transaction) \circ (BTree \bullet Core) \tag{5}$$

$$DB_2 = QE \circ Base \tag{6}$$

$$= (QueryEngine) \circ (BTree \bullet Core) \tag{7}$$

In this example, (1)–(3) describe static compositions resulting in the compound features (i.e., dynamic binding units) *Base*, *QE*, and *TXN*. Equations (4) and (6) represent dynamic compositions of compound features. Hence, a feature such as TRANSACTION is statically bound with respect to its binding unit *TXN* (3) but is dynamically bound with respect to the base program (4).

Note, when combining static and dynamic binding, we have to consider the order in which features are composed. The reason is that composition of feature modules is not necessarily commutative (Apel et al. 2010). For example, when features TRANSACTION and BTREE extend the same method, their composition order may affect program behavior. Hence, when changing the two binding units from (1) and (3) to $Base = Transaction \bullet Core$ and $Log = Logging \bullet Btree$, dynamic composition results in a different program:

$$DB_{1'} = (Logging \bullet Btree) \circ (Transaction \bullet Core). \tag{8}$$

$DB_{1'}$ differs in its behavior from DB_1 if the composition of *Btree* and *Transaction* is not commutative. However, we achieve commutativity when combining static and dynamic binding using special code transformations, as we describe in Sect. 3.4.3.

Feature models for compound features After static composition, it may be necessary to reason about the remaining dynamic variability, which is ideally done with feature model. For example, it is easier to analyze dynamic variability (e.g., to check which configurations are valid) when we have a feature model that includes compound features and that corresponds to the variability of the prebound SPL. This feature model is also reified at run time for validating a configuration before dynamic composition (e.g., to safely reconfigure a running system).

The combination of static binding and dynamic binding can be seen as a *staged configuration* process, as described by Czarnecki et al. (2004). In our case, we have a two step configuration process, in which we first bind features statically and then compose binding units dynamically. We represent this configuration process by transforming the feature model accordingly. In contrast to the model transformations for staged configuration described by Czarnecki et al. (2004), we allow arbitrary configuration steps that are represented by constraints (Classen et al. 2009; Rosenmüller and Siegmund 2010). Composition of multiple features into a compound feature is thus represented by an equivalence constraint between the merged features. That is, when a user selects one of the features of a compound feature, she has to select the other features as well. In Fig. 9b, we depict an example for the

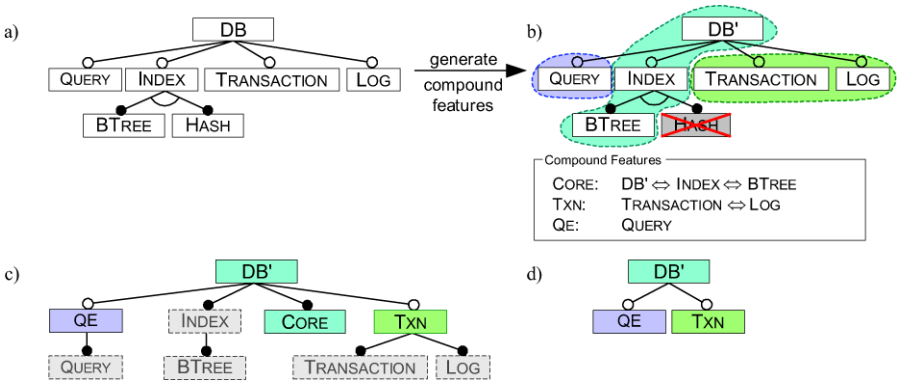

Fig. 9 Transformation of the feature model of a DBMS (**a**) when generating compound features CORE, TXN, and QE. In (**b**), constraints have been added to represent the merge operation. The compound features are added to the feature model in (**c**) and a refactoring is used to include the equivalence constraints. Mandatory subfeatures of the compound features that do not provide variability have been removed in (**d**)

static transformation of *DB* into *DB′* (cf. Fig. 8). We represent compound feature TXN by constraint TRANSACTION ⇔ LOG. Furthermore, we remove feature HASH, which was not selected for composition. The resulting feature model represents the dynamic variability. It forces a user to either select all merged features of a binding unit or to select none of them. For example, features TRANSACTION and LOG can only be selected in combination. However, this feature model is rather complex compared to the actual variability. Furthermore, it does not explicitly show the new compound features which may be needed for dynamic configuration.

We reduced the complexity of the feature model by adding the compound features and by refactoring the model. The resulting feature model is depicted in Fig. 9c. In the following, we describe the required refactorings. Details of the refactoring steps can be derived from Alves et al. (2006).

1. In a first step, we remove dead features that cannot be selected. In our example, this means that feature HASH is removed as it is an alternative to BTREE and cannot be selected. When removing a dead feature, we also remove it from existing constraints to other features (e.g., replacing it with *false* in boolean constraints).
2. We remove a feature from the equivalence constraint of a compound feature if it is the ancestor of one of the other features of the constraint. In turn, we have to mark the features on the path between both in the feature model as mandatory. For example, we mark features INDEX and BTREE as mandatory and remove the equivalence constraint for compound feature CORE. Mandatory features that have been part of an alternative group must always be selected. Hence, the remaining features of the group are dead and have already been removed in step 1.
3. In steps 3–5, we add the compound features and restructure the feature diagram. First, we create a new feature for each generated compound feature (e.g., feature QE in Fig. 9c). Each compound feature replaces one of the merged features. Usually, the compound feature should replace the feature that is nearest to the root. The replaced feature is added as a mandatory child since both have to be

selected at the same time. For example, we insert compound feature QE above QUERY. If one of the merged features is the root of the tree, the compound feature may also be added as a child of the root to avoid a different name for the root (cf. feature CORE in Fig. 9c).

4. Other merged features including their entire subtrees are moved to the corresponding compound feature as mandatory child features (e.g., feature LOG in Fig. 9c). Additional constraints are added to maintain the relationships between the moved features and their former parent features and siblings. In our example, we create the constraint LOG $\Rightarrow$ DB$'$ because feature TXN was added as a parent of TRANSACTION.
5. Finally, we remove constraints that are not needed. Since the merged features are mandatory children of their compound feature, we remove the remaining equivalence constraints that have been used to represent merged features, as described in step 2. Furthermore, we can remove some constraints that have been added in step 4. For example, we remove constraint LOG $\Rightarrow$ DB$'$ because DB$'$ is an ancestor of LOG.

After refactoring the feature model, the remaining variability is easier to recognize because it is not hidden in constraints. The merged features can be removed from the feature model or tool support can be used to suppress visualization of mandatory features. When removing features, their constraints have to be maintained: we have to update all constraints by replacing the removed features with their compound feature. However, the original merged features are still needed for further operations on the feature model. For example, defined independently constraints (e.g., by a third party) may reference the original features.

3.4.3 Implementation: generating binding units

As proof of concept, we implemented our approach on top of the FeatureC++ compiler. In the following, we give a short overview of the code transformations used to combine both types of composition at the class level.

When generating dynamic binding units, the FeatureC++ compiler transforms a class (defined as several class refinements) of an SPL into dynamically composable class fragments. The generated fragments correspond to the binding units that cut accross the class. They are generated from the base class and its refinements in two steps: First, we merge refinements belonging to features of the same binding unit into a single class (static composition) and, second, we generate code for dynamic binding of composed classes using the decorator pattern, as described for dynamic composition (cf. Sect. 3.3). Hence, we do not generate a decorator per refinement, but we group the refinements of a binding unit for each class in a single decorator. In Fig. 10, we show an example for the generated classes of the binding units of *DB*$'$ for class `DB` (cf. Fig. 8). The dynamically composable class `DB` consists of an interface (`DB`), an abstract decorator (`DB_Decorator`), and three concrete decorators (`DB_Base`, `DB_QE`, and `DB_TXN`). Code of multiple refinements is statically composed into concrete decorators `DB_Base`, `DB_QE`, and `DB_TXN`. For example, we merge refinements defined in modules CORE and B-TREE of class `DB` into decorator

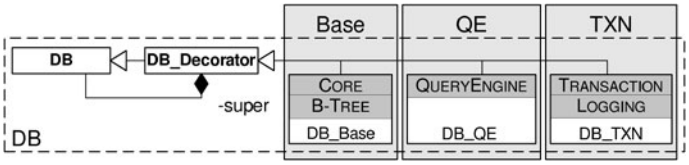

Fig. 10 Compound class DB (*dashed box*) after static composition and transformations to enable dynamic binding. Generated decorators are shown as *white boxes* within light-gray binding units. Code of refinements is shown as *gray boxes*

Feature CORE

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

Feature LOGGING

```
4  refines class DB {
5      bool Put(Key& key, Value& val) {
6          ... //logging-specific code
7          return super::Put(key,val);
8      };
9  };
```

Feature TRANSACTION

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

Fig. 11 FeatureC++ source code of class DB with method Put refined by two features

DB_Base. The decorators are combined at run time according to the selected dynamic binding units. For example, we have to compose DB_Base and DB_TXN to yield DB_1 of Fig. 8.

The code transformations are basically a combination of the transformations described for pure static and dynamic binding. However, they differ in several ways and we present the two most important differences next. In the first example, we demonstrate how to attain commutativity of class refinements. In the second example, we describe how the SPL context is stored in generated classes.

Commutativity of method refinements Since the application of method refinements is usually not commutative, we have to ensure that the application order of method refinements does not change when combining static and dynamic extensions of the same class. An example is shown in Fig. 11: Method Put of class DB is extended in features LOGGING and TRANSACTION. Both method extensions have to be executed bottom-up: first, the transaction code has to be executed (Line 12) and afterward the logging code (Line 6). If we statically compose the CORE implementation and feature

Binding Unit BASE

```
1  class DB_Base {
2      bool Put_Core(Key& key, Value& val) { ... }
3
4      bool Put_hook(Key& key, Value& val) {
5          return Put_Core(key,val);
6      }
7
8      bool Put(Key& key, Value& val) {
9          ... //transaction-specific code
10         return Put_hook(key,val);
11     };
12 };
```

Binding Unit TXN

```
13 class DB_Logging {
14     bool Put_hook(Key& key, Value& val) {
15         ... //logging-specific code
16         return super->Put_hook(key,val);
17     };
18 };
```

Fig. 12 Generated C++ code of class DB with a hook for method refinement

TRANSACTION into a single binding unit and feature LOGGING into a different binding unit, then dynamic composition of the binding units changes the execution order of the method refinements. This results in an invalid program because the transaction code is executed after the logging code.

To avoid this, we generate hook methods (Apel et al. 2008a), as shown in the generated code in Fig. 12. For example, we generate method `Put_hook` (Lines 4–6), which is called instead of method `Put_Core` (Line 10). The hook is overridden by feature LOGGING to execute the logging specific code before executing the extended method (Line 16).

Storing SPL context Class instantiation in a dynamically composed program requires to create an object that corresponds to the configuration of a concrete SPL instance. The instance defines which decorators to use when creating the new object. Because there may be more than one active SPL instance within a program, we need to know which instance to use when creating an object. For that reason, we store a reference to the corresponding SPL instance within each object. For example, when creating an instance of class DB (cf. Fig. 10), the SPL instance DB_1 defines the required binding units (BASE and TXN) and thus the configuration of class DB.

For statically composed classes, this information is not needed because the type of a class is determined statically and does not change according to a dynamically changing SPL instance. For example, the type of class `QueryProcessor` in Fig. 2 does not change if features QUERYENGINE and TRANSACTION are part of the same binding unit. Hence, there is no runtime variability for such a class and we do not need an SPL instance for creating objects of the class. We thus evaluate whether a class (directly or indirectly) creates instances of dynamically composed classes or

not; if it does, it has to store a reference to the corresponding SPL. For example, if class `QueryProcessor` is statically composed (because it is only part of a single binding unit), it has to store a reference to its SPL instance only if it creates objects of other dynamically composed classes.

Since the SPL reference is only needed when there are multiple instances of the same SPL, we further optimize the generated objects when only a single instance is used. In this case, we use a single global reference to the SPL instance to avoid the additional SPL instance pointer per object. A user can make this decision at deployment time by choosing between alternative code transformations.

3.4.4 Summary

When combining static and dynamic composition, the compiler merges multiple features into a binding unit and generates code to support composition of binding units at run time. Hence, a dynamically bound feature is statically composed with features of the same binding unit; it may even use only static binding of its classes and class refinements with classes of other features of the binding unit. To reason about the resulting dynamic variability, we apply the static composition process also to an SPL's feature model. We transform the feature model according to the generated binding units. The result is a feature model that provides only dynamic variability. It can be used to analyze dynamic binding before generating binding units, to configure an SPL at run time (e.g., for a run time adaptation approach), and to verify a dynamic composition before using it. Overall, our approach integrates static and dynamic binding at the modeling level as well as at the implementation level. Hence, it provides a foundation for an SPL development process that is independent of the supported binding time.

4 Case studies and evaluation

By means of two case studies, we demonstrate the applicability of our approach. We measure the influence of different sizes and different numbers of dynamic binding units on resource consumption. For our evaluation, we use two product lines that have been developed at the University of Magdeburg. The first SPL is FAME-DBMS, a DBMS product line for resource-constrained environments (Rosenmüller et al. 2009). The second SPL is NanoMail, a customizable e-mail client. The source code of both product lines is available on the web.[6] We present the results for both SPLs and discuss the reasons for the characteristics we observed.

4.1 Defining binding units

FAME-DBMS FAME-DBMS is an embedded DBMS (i.e., it is embedded into an application as a library). It was developed for devices with limited resources using static feature binding. For compositional flexibility and to reduce the functional overhead,

[6] http://wwwiti.cs.uni-magdeburg.de/~rosenmue/dynamic/.

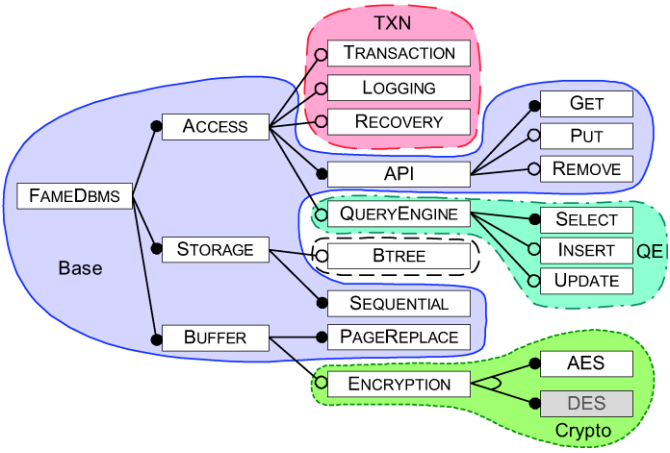

Fig. 13 Feature diagram of FAME-DBMS with binding units Base, TXN, Btree, QE, and Crypto. Binding unit Crypto consists either of feature AES or DES. In our evaluation, we use feature AES

we use dynamic binding. In Fig. 13, we depict an excerpt of the feature model of FAME-DBMS and the binding units used in our evaluation. We show only features that are relevant for our case study and omit features that are always bound statically such as operating-system-related features. In its current version, FAME-DBMS consists of 56 features with 12 400 lines of code (LOC).

For analyzing the influence of dynamic binding on resource consumption, we compare different variants of FAME-DBMS that use the same 44 features, but we organized the features in different binding units. The selection of features per binding unit is shown in Fig. 13. It corresponds to configuration 5 in the following analysis. We describe the rationale behind the definition of the sample binding units in the following overview:

- Binding unit BASE represents a basic DBMS that consists of an API for storing and retrieving data. It can be used without additional binding units and provides high performance due to pure static binding.
- Binding unit TXN provides transactional access to the database. Since features TRANSACTION and RECOVERY require feature LOGGING, we merge all three features into a single binding unit.
- QE is a customizable query engine that supports a subset of SQL by statically composing only the required SQL features. In our implementation, dynamic composition of SQL features is hard to achieve. The reason is that we statically compose the SQL grammar from multiple features. We then generate the SQL parser from this composed grammar at compile time. This demonstrates that pure dynamic binding is not always possible without increasing the development effort significantly.
- CRYPTO is a binding unit for data encryption and decryption. Customization of ciphers is done statically by choosing the encryption algorithms (e.g., AES or DES). This means that we can exchange the encryption algorithm within the binding unit without modifying the remaining DBMS. We may also provide two different CRYPTO binding units, one with feature AES and one with DES. Moreover, a

customer may provide an own encryption algorithm. Defining one binding unit for the DES and AES features would also be possible, but in our case the ENCRYPTION feature abstracts from implementation details of the algorithm resulting in a small and uniform interface.
– Finally, binding unit BTREE provides efficient data access via a B^+tree index structure. In a large DBMS, there may be a number of different alternative index structures. Using a single binding unit per index structure allows us to activate only those index structures that are needed for efficiently accessing the data (i.e., depending on the work load).

NanoMail NanoMail is an e-mail client SPL with 25 features and 6 200 LOC. It comprises different e-mail applications, from a simple MailNotify application that only notifies a user if there is unread mail, up to a full mail client with mail storage in a database. Similarly to FAME-DBMS, we compare variants with equal functionality (using 23 features) and varying binding units. For dynamic binding, we defined the binding units DB and CLAMAV, which provide database storage and virus filtering. Furthermore, for analyzing the impact of fine-grained dynamic customization, we provide e-mail filters that are used like plugins. Each filter is loaded as a single binding unit and users can add as many filters as needed. To analyze the influence of a large number of binding units, we generate several mail filters and measure the effect on startup time.

4.2 Resource consumption

In the following, we analyze the resource consumption of different FAME-DBMS and NanoMail variants depending on the binding units used. We compare binary size, working memory usage, and performance of a varying number of binding units but we use always the same features.[7] Our aim is to identify how to combine static and dynamic binding to optimize a program with respect to functional and compositional overhead.

In Figs. 14–19, we depict the results of our analysis for five configurations of FAME-DBMS and three configurations of NanoMail. In configuration 1, all features are statically bound and compiled as a single binary. In each of the configurations 2–5 an additional binding unit (e.g., QE, TXN, CRYPTO, BTREE for FAME-DBMS) is extracted from the base binding unit and compiled as a distinct dynamically linked library (DLL). In the following, we analyze binary size, memory consumption, and performance of both SPLs. We distinguish between compositional and functional overhead (cf. Sect. 2) for each analyzed property.

[7]For our evaluation, we used an Intel Core 2 system with 2.4 GHz and Windows XP. For compilation, we used the Microsoft C/C++ compiler v13.10.3077 and Incremental Linker v7.10.3077 (Visual C++ 2003). We used compiler optimization flag /O2 (i.e., /Og/Oi/Ot/Oy/Ob2/GS/GF/Gy). We linked dynamically against Microsoft's C++ run time library and removed unreferenced functions and data with linker flag /OPT:REF.

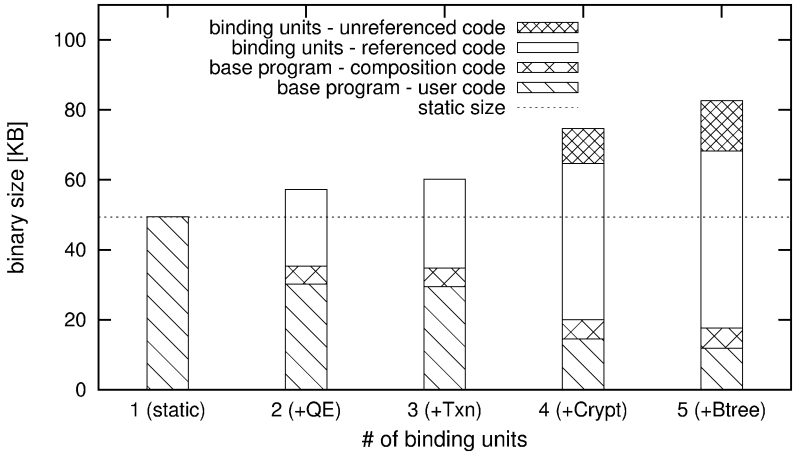

Fig. 14 Binary size (base program and dynamic binding units) of five FAME-DBMS variants with an equal feature selection and an increasing number of binding units

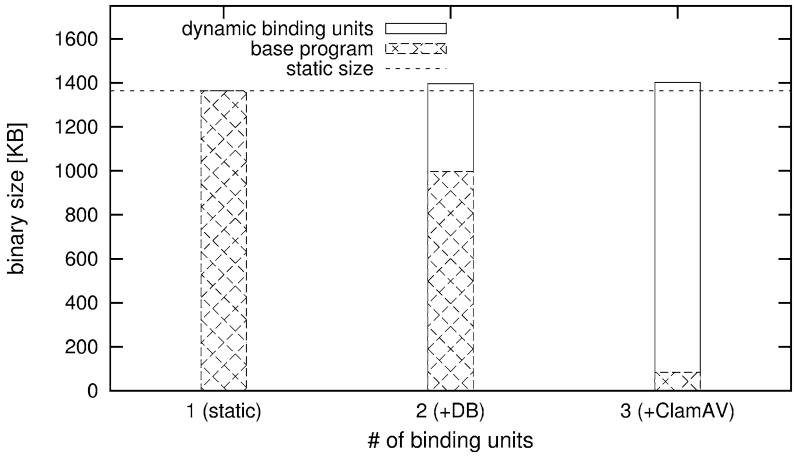

Fig. 15 Binary size of three variants of NanoMail with varying sets of binding units

4.2.1 Binary size

By means of the binary size of FAME-DBMS, we first describe how we calculate the functional and compositional overhead. Since the functional overhead depends on the features actually used, we provide numbers for the *maximal possible* functional overhead. That is, we compare a static variant including all features with the minimal dynamic variant without additional binding units. The binary sizes of the configurations 1–5 of FAME-DBMS are shown in Fig. 14. The values represent executable code and static data stored in the binary files. They do not include other libraries. For configuration 1, we generated a single binding unit including all features and five binding units for configuration 5. From configuration 1, we can only derive a single variant

with a binary size of about 50 KB. Comparing configurations 1 and 5, we observe the following compositional and functional overhead:

- Comparing a complete variant of configuration 5 (83 KB) with configuration 1 (49 KB), we observe a *compositional* overhead of about 40% (83 KB − 49 KB = 34 KB). That is, 40% of the code from configuration 5 is required for dynamic binding.
- Comparing configuration 1 (49 KB) and the smallest variant of configuration 5 (18 KB), we observe a maximal possible *functional* overhead of about 64% for configuration 1 (49 KB − 18 KB = 31 KB). That is, up to 64% of the code of configuration 1 may not be used for a particular task (e.g., basic data storage and retrieval without using SQL queries and other features). The overhead depends on the number of features that are actually used at a particular point in time. It is zero when all features are really in use. This underlines that a configuration highly depends on the application scenario. To reduce the binary size, we have to avoid any features that are not used and reduce dynamic binding to a minimum.

In our case studies, we observe an increasing compositional overhead for an increasing number of binding units. Especially when a binding unit extends many classes the effect is very strong. It is quite strong for FAME-DBMS (up to 40%, cf. Fig. 14) and very weak for NanoMail (<4%).

The high relative overhead of 40% for FAME-DBMS is mainly caused by its small binary size; the absolute overhead is 33.2 KB. The composition code makes up 21 KB (25% of the program size): About 5 KB generic code for dynamic binding (i.e., for loading and composing binding units; *base program—composition code* in Fig. 14) and additionally between 3 KB and 5 KB overhead per binding unit (i.e., binding unit specific composition code). The remaining overhead of 12 KB (15% of the program size) is caused by the binding units CRYPTO (9 KB) and BTREE (3 KB). Reasons are missing compiler and linker optimizations when dynamic binding is used. In Fig. 14, we depict this unused code as *binding units—unreferenced code*. For example, a method for calculating a hash sum with a binary size of 7.5 KB is not used in our FAME-DBMS benchmark application. The linker removes the method from the statically composed variant because it is never called. The same method cannot be removed from binding unit CRYPTO because the compiler does not know whether it is required by another binding unit or not. Hence, dynamic binding may cause a functional overhead as well. This overhead is not caused by entire unused features but by unreferenced methods.

The possible functional overhead in static variants of both SPLs is very high (64%–94%). The reason is that a large fraction of the binary code belongs to optional features. Increasing the use of dynamic binding usually reduces this overhead. However, also insufficient customizability due to large binding units can cause a functional overhead when not all features of a binding unit are used.

Both kinds of overhead can be reduced by adjusting the binding units. That is, a stakeholder has to analyze the functional and compositional overhead per application scenario to find the optimal tradeoff. When always using many of the binding units, the benefit of dynamic binding with respect to resource consumption decreases. For example, the binding units TXN and BTREE in FAME-DBMS cannot significantly

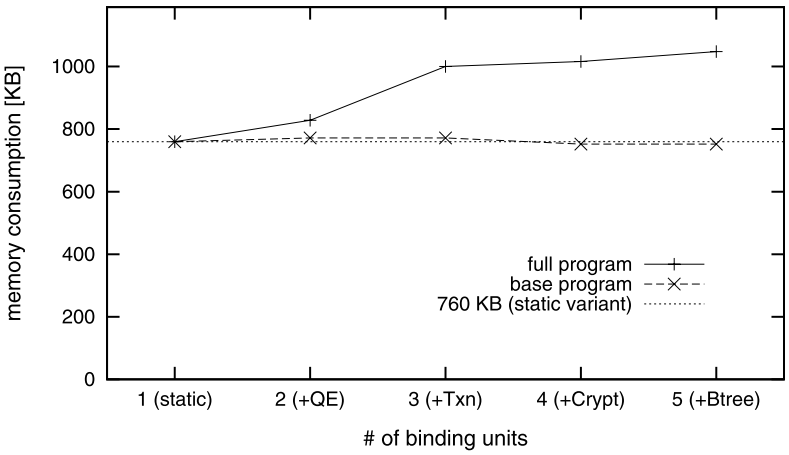

Fig. 16 Comparison of working memory usage of five FAME-DBMS variants with an equal feature selection and an increasing number of binding units. *Full program* variants include all dynamic binding units. *Base program* variants use only the base program without loading additional binding units

reduce the functional overhead but they increase the compositional overhead significantly. The size of the base program is nearly the same in the configurations 4 and 5, but the additional binding unit BTREE increases the overall size by 12%. That is, if we bind the features of the binding units TXN and BTREE statically (i.e., removing configurations 3 and 5), we do not cause a major functional overhead but can reduce the compositional overhead significantly. Hence, a stakeholder has to decide whether this flexibility is really needed by taking the resulting compositional overhead into account.

4.2.2 Memory usage

The memory usage of a program depends on allocated memory but also on the size of the binary program code that is loaded into memory. For FAME-DBMS, we could not measure any functional overhead of allocated memory because the memory is needed mainly for the data buffer of the DBMS, which is independent of the feature selection. Further features do not allocate a significant amount of additional memory. The functional overhead thus only depends on the binary program size and dynamic binding cannot reduce the memory consumption (cf. Fig. 16). In NanoMail, memory allocation causes an additional memory consumption between 1.3 MB and 9.8 MB depending on the binding units used. That is, unused features allocate memory and cause a large functional overhead. On the contrary, the varying binary size only has a small effect on memory consumption in NanoMail (about 28%, cf. Fig. 17).

The compositional overhead of allocated memory especially increases if a program creates a large number of small objects that are dynamically composed. The reason is that the size of a class instance increases for each binding unit that crosscuts the class. This overhead is very high for small objects. The size S_{dyn} and the overhead

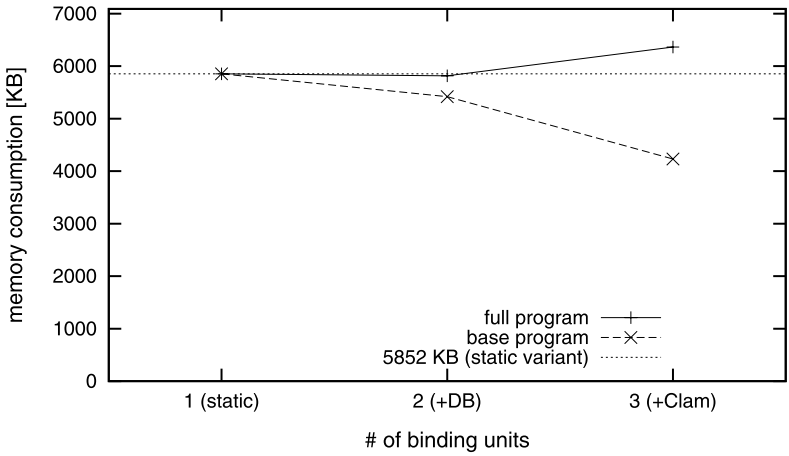

Fig. 17 Consumed working memory in NanoMail for variants with different binding units

O_{dyn} of a dynamically bound object can be calculated with the following formula:

$$S_{dyn} = S_{data} + O_{dyn} \tag{9}$$

$$O_{dyn} = 12(n_{BU} + 1) \tag{10}$$

S_{data} is the data size (i.e., the size of the object with static binding). n_{BU} is the number of *loaded* binding units that crosscut the class. The constant 12 represents the number of bytes a binding unit requires to store a self pointer, a super pointer, and a pointer to its virtual function table (each pointer has a size of 4 byte). The constant 1 represents the additional proxy that we use to enable reconfiguration at run time. The proxy could be removed for SPLs that are not reconfigured once they are instantiated. Overall, the size of an object increases by 12 byte to enable dynamic binding and linearly increases by 12 byte for each additional binding unit.

Large binding units increase the functional overhead only if a small number of their features is actually used. The compositional overhead in FAME-DBMS is more important than the functional overhead; the opposite is true for NanoMail. The differences between the SPLs show that there is no general solution and there is space for domain-specific optimizations.

A binding unit also increases memory consumption due to the compositional overhead of its binary size (cf. Sect. 4.2.1) because the executable code is loaded into memory. We analyzed the overhead for loading a large number of binding units by adding several e-mail filters to NanoMail (up to 60 mail filters).[8] The results are shown in Fig. 18. Besides a general overhead for dynamic binding (transition from 0 to 1 filters), we observe a linear increase of 21 KB per filter (i.e., per binding unit). For binding units that consume a small amount of memory, this is a large overhead. By generating one binding unit for multiple filters (i.e., merging the filters), this overhead can be avoided. For binding units that consume a large amount of memory, such

[8] We generated empty filter stubs to measure only the overhead for dynamic composition.

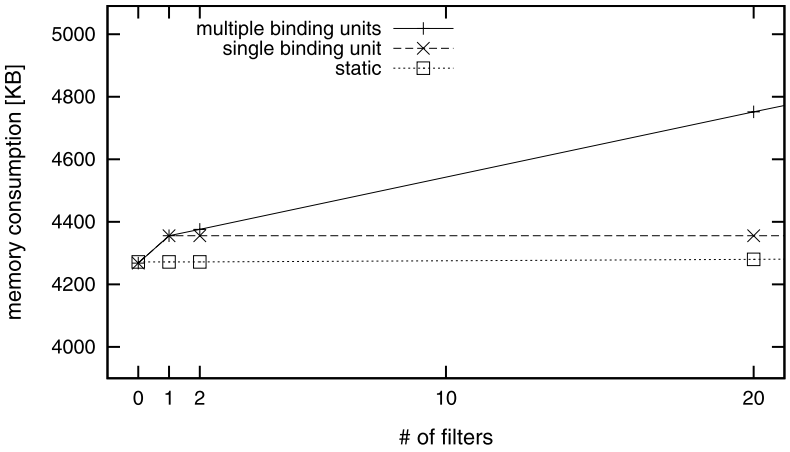

Fig. 18 Consumed working memory of NanoMail with an increasing number of mail filters with static composition, dynamic composition with a single binding unit for all filters, and pure dynamic composition (i.e., one binding unit for each filter)

as the virus filter in NanoMail, dynamic binding causes an acceptable overhead of only about 4% compared to the memory consumption of the binding unit.

4.2.3 Performance

We measured the performance of FAME-DBMS using a benchmark for reading and writing data.[9] As shown in Fig. 19, the performance decreases with an increasing number of binding units. Comparing dynamic variants with pure static binding, we observe a performance reduction between 5% (2 binding units) and 28% (5 binding units). The reason for this increasing compositional overhead are method inlinings that could not be applied and more indirections for method calls compared to static variants. Both are caused by generated code for dynamic binding: composition of classes at run time is achieved with virtual methods in decorators, which add an indirection and hinder inlining of method extensions.

In FAME-DBMS, 100% of method refinements are inlined when using static binding. This decreases to about 95% for configuration 2 and further to 86% for configuration 5. Each method refinement that is not inlined is replaced by a virtual method and thus decreases performance. Binding unit BTREE substantially increases the overhead (cf. configuration 5 in Fig. 19). The reason is that it refines methods that are invoked multiple times for a single read or write operation. Hence, we should create only a distinct binding unit for the Btree if this flexibility is really needed (e.g., when we have to decide at run time which kind of index to use).

Static and dynamic binding may also affect the startup time of a program for loading binary code from DLLs and for initialization of unused code. Due to the fairly

[9] We used random key-value-pairs for reading and writing 10.000 records of type string via the B-Tree index.

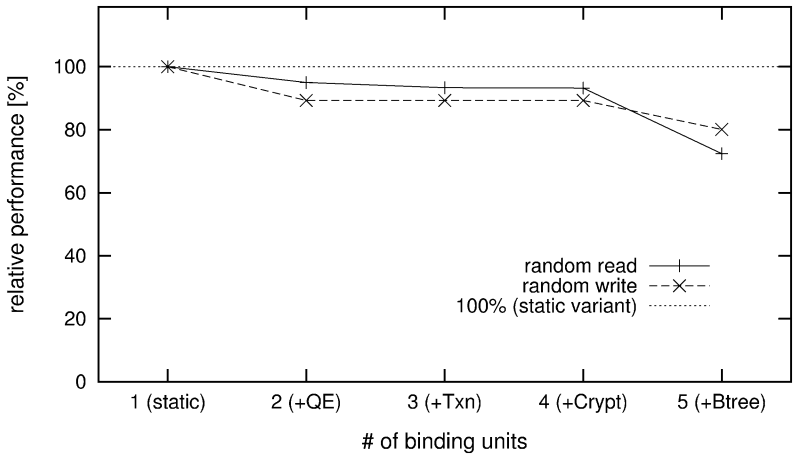

Fig. 19 Comparison of benchmarks for reading and writing of different variants of FAME-DBMS. The performance relative to static binding is shown. 100% means about 3.0 Mio queries/s for reading and 0.8 Mio queries/s for writing

small binary size of binding units, we observed only an slightly increased startup time. The compositional overhead for loading binding units is about 30 ms per additional binding unit. We observed a functional overhead (intitialization code of features) of about 2 s for the largest binding unit in NanoMail (the CLAMAV virus filter). Hence, the compositional overhead with respect to program startup is very small and can be ignored in many application scenarios. In contrast, the functional overhead for initialization of a binding unit may be important for application scenarios that require to restart a program frequently.

To summarize, the influence of dynamic binding is quite high when a feature refines frequently called methods. This is may be caused by a high number of method extensions (e.g., in many binding units), but it may also be caused by a few refinements of performance critical methods as shown for the Btree in FAME-DBMS. Again, the best size for a binding unit has to be determined per SPL and application scenario. Merging binding units can remove dynamic method refinements. The load time of a program can only be reduced significantly if the execution of complex initialization code can be avoided or if large parts of a program do not have to be loaded at startup. The number of binding units is usually not important. For example, 30 binding units result in an overhead of about one second for starting the programs of our case study.

5 Discussion

In this section, we discuss the results of our evaluation and analyze how customizability and SPL development is influenced by our approach. Finally, we derive a guideline for building SPLs that support static and dynamic binding.

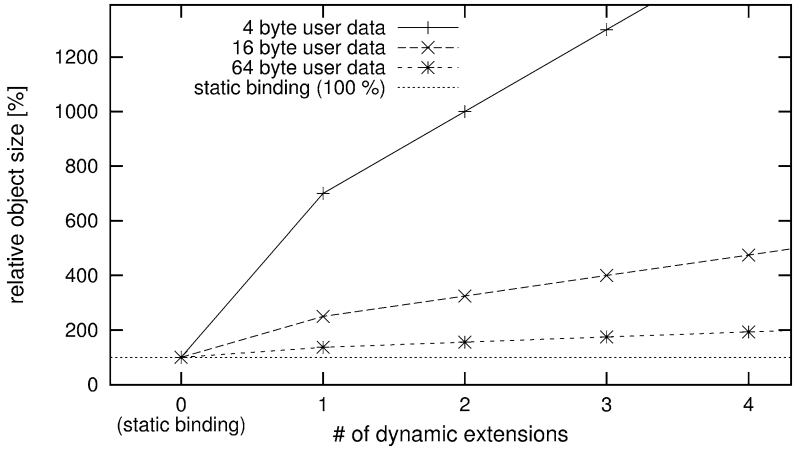

Fig. 20 Relative size of three objects with a data size of 4, 16, and 64 bytes for an increasing number of dynamic extensions (i.e., crosscutting binding units)

5.1 Resource consumption

Our evaluation has shown that, depending on the binding time, a compositional and functional overhead occurs in a running program with regard to binary size, memory consumption, and performance. The compositional overhead caused by a binding unit depends on its entanglement with other binding units. That is, method calls across the boundary of a binding unit (i.e., via its interface) introduce an execution time overhead. The interface of a binding unit consists of virtual methods to enable dynamic binding. This hinders method inlining, introduces indirections, and increases the size of generated code as well as the size of objects in a running program. Hence, a binding unit should contain feature sets that are used in combination. For example, the effect on memory consumption is very high when allocating a large number of small objects. In Fig. 20, we depict the computed relative size (cf. (9)) of three different objects with 4, 16, and 64 bytes user data with an increasing number of dynamic extensions (i.e., dynamic binding units that crosscut the object). For an object with 4 byte user data, two dynamic extensions increase the object size by a factor of 10. If such objects are the main cause of memory consumption of a program then the memory consumption also increases by a factor of 10. For larger objects, this effect is much smaller. Combining static and dynamic binding reduces the number of dynamic binding units and can thus highly decrease the memory consumption.

However, large binding units introduce a potential functional overhead due to features that are not used. Splitting binding units can reduce the functional overhead, but we have shown that this effect can be smaller than the introduced compositional overhead. Furthermore, we have shown that dynamic binding may also introduce an overhead due to unused methods that can be removed by the linker when using static binding. An advantage of our approach is that it allows a programmer to find a balance between compositional and functional overhead that is suitable for her needs.

As shown in Fig. 21, our approach provides pure static and pure dynamic composition (lower left and upper right points) as well as all combinations with varying sets

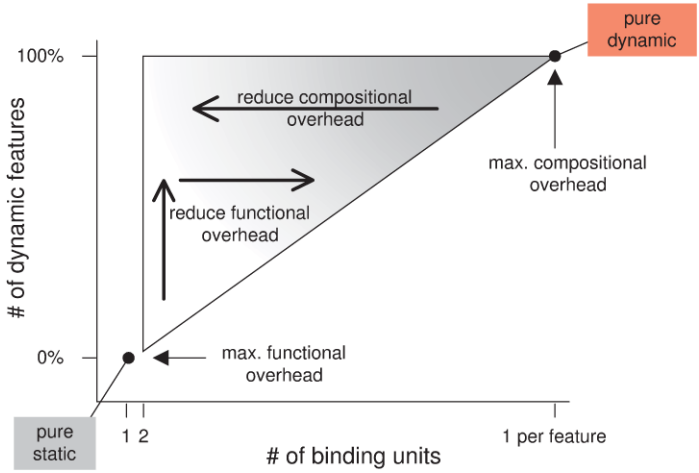

Fig. 21 Combining static and dynamic binding to support a varying number and size of binding units

of binding units (shown as triangle). When creating binding units, the compositional overhead can be reduced for a constant number of dynamically bound features by increasing the number of features per binding unit (arrow in upper part of Fig. 21). The functional overhead can be reduced in two ways (lower left arrows): On the one hand, increasing the number of dynamically bound features (i.e., moving features from the base program into a binding unit) reduces the size of the base program. On the other hand, increasing the number of dynamic binding units (i.e., splitting the binding units) reduces the size of each binding unit.

Optimizing resource consumption Our approach and its current implementation does not provide an optimized solution for every application scenario. To optimize memory consumption and execution time, we already provide different code transformations to generate applications that use a single SPL instance only or programs that use multiple instances of an SPL (cf. Sect. 3.4.3). However, we can further optimize the code transformation process to reduce the compositional overhead caused by dynamic binding. For example, we can reduce memory consumption by allocating a dynamically composed object in a single block of memory instead of multiple blocks—one for each decorator. This allows us to reduce the object size by removing the `super` and `self` pointers. Instead, we can compute the memory addresses of the super object and the compound object for each compound class at instantiation time, as it is also done for inheritance by C++ compilers (Lippman 1996). Hence, the memory consumption of small dynamically composed objects (e.g., as shown in Fig. 20) could be reduced to about one third. However, this solution is only better suited for SPL configuration at load time. Using this approach for reconfiguration at run time (e.g., adding a new refinement to an already existing object) means that we have to reallocate the whole object when its size increases (i.e., when a new feature is loaded). This may highly increase the time required for adaptation. Hence, such optimizations are usually well suited for a particular application scenarios only.

5.2 Customizability and SPL development

Granularity and flexibility Due to the dynamic binding capabilities of FeatureC++, a developer can achieve extensibility of a program after deployment. Additionally, features can be bound statically, which supports fine-grained extensions without increasing the execution time. For example, the B-Tree in FAME-DBMS is built from many small features (e.g., features for write support) that can be statically configured. This allows us to achieve a performance and memory consumption comparable to static binding with the C preprocessor, as we have shown in previous work (Rosenmüller et al. 2009). Fine-grained extensions and dynamic binding are opposite optimization goals with respect to performance and memory consumption. The more fine-grained the extensions are, the more memory and computing power is required for dynamic binding (cf. Fig. 20). Our approach allows a programmer to combine both binding times as needed per application scenario.

Reuse The combination of both binding times increases reuse possibilities in different application scenarios. For example, we can statically bind all features of FAME-DBMS for deeply embedded devices and support dynamic binding for other platforms. Furthermore, binding time flexibility enables reuse of features across different SPLs that use different binding times. For example, we can reuse a feature that implements a communication protocol in an e-mail client SPL that uses dynamic binding and also in an e-mail server that uses static binding.

Crosscutting features With our approach, static binding can also be used for crosscutting features that are spread across multiple dynamic binding units. These features are usually implemented with preprocessors (Griss 2000) or design patterns (Mezini et al. 2000; Zdun 2004). Adding or removing such features is possible by rebuilding the affected binding units. For example, in FAME-DBMS, WRITESUPPORT is a crosscutting feature that affects several binding units such as the query engine, indexes, etc. Using FeatureC++, we can add or remove this feature and have to rebuild only the affected binding units.

Development and maintenance Using a single mechanism for implementing features (i.e., feature modules) also simplifies SPL development. A programmer may combine FeatureC++ with other variability mechanisms (design patterns, macros, #ifdefs, etc.) or may replace other mechanisms by feature modules. Especially the use of #ifdefs can be reduced to improve the comprehensibility of source code. Binding time flexibility can also simplify maintenance of an SPL. For example, dynamic binding can be replaced temporarily by static binding for debugging purposes to avoid the complexity of dynamic binding. Finally, the presented approach can also be applied if initially only static binding is required. In this case, it simplifies adoption of dynamic binding (e.g., in later versions of an SPL, as shown for FAME-DBMS).

5.3 A guideline for defining binding units

When configuring an SPL for static and dynamic binding, we have to answer two questions: Which features have to be bound dynamically? Which dynamically bound

features should be composed into the same binding unit? With our approach, a domain expert can decide this per application scenario. Static binding does not exhibit any compositional overhead. It is usually the best choice if extensibility after deployment or at run time is not required. The remaining challenge for a domain expert is to find proper binding units for dynamically bound features to provide the required flexibility while minimizing the overhead. Therefore, resource consumption of different feature combinations has to be analyzed, which means a high effort and may be impractical. The following rules can be used to find good feature combinations for binding units more easily:

1. As a simple rule, a large number as well as a large size of binding units should be avoided because the first increases the compositional overhead and the latter increases the functional overhead. However, as depicted in Fig. 21, this cannot be a fixed rule because reducing one overhead may increase the other.
2. Analyzing the feature model helps to find features that should be combined in a binding unit. For example, a feature and all its children should usually be part of the same binding unit. Similarly, a *requires* constraint between two features indicates that the features are used in combination and may also be combined into the same binding unit.
3. Furthermore, we can analyze the source code of features. A high degree of coupling between features indicates which features are commonly used together (Apel and Beyer 2011). Hence, it can be beneficial to merge them into a single binding unit. Crosscutting features should be bound statically if possible. An automated analysis of coupling and cohesion could be used to provide an initial assignment of dynamically bound features to binding units.
4. Implementation knowledge can be used to find features and methods that are important with respect to performance and memory consumption. For example, frequently called *hot spot* methods should ideally be bound statically. If this is not possible, they should be defined in a single binding unit only. This causes the method to be bound dynamically but avoids a decomposition of the method into multiple fragments. Similarly, when allocating a large number of small objects (such as list elements), the corresponding class should be defined in a single binding unit.

To further reduce the overhead of a program, different optimizations of binding units are possible. For example, *overlapping binding units* (i.e., binding units that use an overlapping set of features) can be created to provide binding units with a small interface or to reduce the number of binding units. Another optimization is to *split* or *merge* binding units when the requirements have changed over time or when an analysis at runtime has identified how the binding units are actually used. For example, binding units that are often or always used in combination can be merged into a single binding unit without changing the source code.

6 Related work

There are approaches for software composition that employ different techniques or paradigms to support different binding times. For example, CaesarJ (Aracic et al.

2006) supports static composition based on virtual classes and dynamic deployment of aspects. Object Teams support dynamic binding of *teams*, which can be used to represent features of an SPL (Hundt et al. 2007). Furthermore, *activation teams* are statically instantiated. These in turn activate other teams at run time. Both approaches require to know the binding time of an implementation unit at design time. In contrast, we can choose the binding time at deployment time to enable reuse of source code even when using different binding times.

Zdun et al. introduce transitive mixins to generalize composition of classes and objects (Zdun et al. 2007). The implementation provided in Zdun et al. (2007) is built on top of a dynamic approach that does not support static composition. Chakravarthy et al. provide with Edicts an approach that supports different binding times using design patterns that are applied to a program by means of aspects (Chakravarthy et al. 2008). Configuration is done by switching between Edicts. Czarnecki et al. describe how to parameterize the binding time using C++ templates (Czarnecki and Eisenecker 2000). They provide a configurable binding time (e.g., for class extensions) with a template-based program generator. The OSGi[10] standard also allows static and dynamic composition of components, called *bundles*. However, it is a component-based approach that does not allow a system to be decomposed into fine-grained (crosscutting) features. Other approaches support static and dynamic binding of aspects. AspectC++ supports weaving at run time and compile time for the same aspect (Gilani and Spinczyk 2005). AspectJ supports weaving advice at compile-time, after compile-time, and at load time.[11] PROSE (Nicoara et al. 2008), Steamloom (Bockisch et al. 2004), Hotwave (Villazón et al. 2009), and other AspectJ-based approaches support weaving at run time and may be combined with AspectJ's static weaving. These AOP approaches can be used to support multiple class extensions at the same time as in FOP.

In contrast to FeatureC++, the approaches above do not provide a mechanism for feature composition according to a feature model, which is necessary for validation. Nevertheless, they can be combined with tools that support configuration and static composition of SPLs such as pure::variants (pure-systems GmbH 2004). However, the approaches do not provide a mechanism for configuration at run time that is based on a feature model. Similar to our approach, the mechanisms combine static and dynamic binding. Approaches such as Edicts and AspectC++ can be used to bind a feature statically or dynamically with the base program without changing the implementation. However, they do not provide means to statically merge an arbitrary set of dynamically bound features into a single binding unit and compose the binding units dynamically. One reason is that these approaches do not preserve the execution order of method extensions when mixing both binding times in this way. Applying static binding first and dynamic binding afterwards changes the feature composition order (all static features are bound before dynamic features). This in turn may change the behavior of methods that are refined by statically *and* dynamically bound features (e.g., statically and dynamically weaved aspects that have the same join point). We solve this by generating hook methods, as described in Sect. 3.4.3.

[10] http://www.osgi.org.

[11] http://eclipse.org/aspectj.

Lee et al. suggest to decide before SPL development which features to implement in one component and to combine the resulting components at run time (Lee and Kang 2006). Griss argues that components and novel approaches to software composition should be combined to develop SPLs (Griss 2000). He discusses different approaches that may be used to customize components when the feature selection changes. Our approach goes into the same direction. It uses only a single implementation technique that supports static and dynamic binding. We also think that components have to be planned before SPL development, but the selection of concrete features and component customization has to happen at deployment time.

Our approach is partially based on the Delegation Layers approach (Ostermann 2002), which supports dynamic binding of features. Several other collaboration-based approaches and layered designs such as Jak (Batory et al. 1998), Java Layers (Cardone and Lin 2001), Jiazzi (McDirmid et al. 2001), Mixin Layers (Smaragdakis and Batory 2002), Aspectual Feature Modules (Apel et al. 2008b), Aspectual Collaborations (Lieberherr et al. 2003), and Context-oriented Programming (Hirschfeld et al. 2008) also support either static or dynamic composition. In contrast to these approaches, our approach integrates static and dynamic binding.

FeatureC++ supports composition at run time but is not a full-fledged solution for run time adaptation of SPLs. For example, we do not provide an infrastructure for context dependent activation and deactivation of features. It is not without reason that there is a whole branch of research on run time adaptable SPLs (Hallsteinsen et al. 2008). Nevertheless, solutions for run time adaptation can be built on top of FeatureC++ to combine traditional SPLs with run time adaptable SPLs and to additionally support static customization of components.

7 Conclusion and perspective

We have presented an approach that seamlessly integrates static and dynamic feature binding at implementation and modeling level. Our approach allows developers to statically generate *dynamic binding units* that can be composed at load time or at run time of a program. The approach overcomes limitations of pure static and pure dynamic binding and can be used to replace existing approaches by using only a single implementation mechanism. We provide means to:

- develop the features of an SPL using a single implementation mechanism that is independent of the binding time,
- choose the binding time per feature after development,
- generate dynamic binding units by composing multiple features to optimize resource consumption.

Compared to alternative solutions for implementing static and dynamic feature binding, our approach simplifies several aspects of SPL development. With respect to resource consumption, we found a tradeoff between *functional overhead* caused by static binding and *compositional overhead* caused by dynamic binding. Finding the optimal binding time for the features of an SPL is a difficult task. Varying requirements on flexibility between different application scenarios further complicate the

decision. Our proposal of generating dynamic binding units, allows an SPL developer to choose the binding time per feature even after development and for each application scenario individually.

In future work, we plan to integrate our approach with component-based software development (e.g., to simplify component integration and to avoid dynamic binding of components if it is not required). This means that a binding unit has to clearly define an interface that can be used by other binding units.

Acknowledgements We thank Christian Kästner for comments on earlier drafts of this paper. The work of Marko Rosenmüller is funded by the German Research Foundation (DFG), project number SA 465/34-1. Norbert Siegmund is funded by the 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.

References

Alves, V., Gheyi, R., Massoni, T., Kulesza, U., Borba, P., Lucena, C.: Refactoring product lines. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 201–210. ACM Press, New York (2006)

Anastasopoules, M., Gacek, C.: Implementing product line variabilities. In: Proceedings of the Symposium on Software Reusability (SSR), pp. 109–117. ACM Press, New York (2001)

Apel, S., Beyer, D.: Feature cohesion in software product lines. In: Proceedings of the International Conference on Software Engineering (ICSE) (2011, to appear)

Apel, S., Kästner, C.: An overview of feature-oriented software development. J. Object Technol. **8**(5), 49–84 (2009)

Apel, S., Kästner, C., Batory, D.: Program refactoring using functional aspects. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 161–170. ACM Press, New York (2008a)

Apel, S., Leich, T., Saake, G.: Aspectual feature modules. IEEE Trans. Softw. Eng. **34**(2), 162–180 (2008b)

Apel, S., Lengauer, C., Möller, B., Kästner, C.: An algebraic foundation for automatic feature-based program synthesis. Sci. Comput. Program. **75**(11), 1022–1047 (2010)

Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An overview of CaesarJ. In: Transactions on Aspect-Oriented Software Development I. Lecture Notes in Computer Science, vol. 3880, pp. 135–173. Springer, Berlin (2006)

Batory, D., Lofaso, B., Smaragdakis, Y.: JTS: Tools for implementing domain-specific languages. In: Proceedings of the International Conference on Software Reuse (ICSR), pp. 143–153. IEEE Computer Society, Washington (1998)

Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling step-wise refinement. IEEE Trans. Softw. Eng. **30**(6), 355–371 (2004)

Bockisch, C., Haupt, M., Mezini, M., Ostermann, K.: Virtual machine support for dynamic join points. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 83–92. ACM, New York (2004)

Cardone, R., Lin, C.: Comparing frameworks and layered refinement. In: Proceedings of the International Conference on Software Engineering (ICSE), pp. 285–294. IEEE Computer Society, Washington (2001)

Chakravarthy, V., Regehr, J., Eide, E.: Edicts: Implementing features with flexible binding times. In: Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), pp. 108–119. ACM, New York (2008)

Classen, A., Hubaux, A., Heymans, P.: A formal semantics for multi-level staged configuration. In: Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pp. 51–60 (2009)

Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, Reading (2000)

Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration using feature models. In: Proceedings of the International Software Product Line Conference (SPLC). Lecture Notes in Computer Science, vol. 3154, pp. 266–283. Springer, Berlin (2004)
Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
Gilani, W., Spinczyk, O.: Dynamic aspect weaver family for family-based adaptable systems. In: Proceedings of Net.ObjectDays, pp. 94–109. Gesellschaft für Informatik, Munich (2005)
Griss, M.L.: Implementing product-line features with component reuse. In: Proceedings of the International Conference on Software Reuse (ICSR). Lecture Notes in Computer Science, vol. 1844, pp. 137–152. Springer, Berlin (2000)
Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic software product lines. Computer **41**(4), 93–95 (2008)
Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. J. Object Technol. **7**(3), 125–151 (2008)
Hundt, C., Mehner, K., Pfeiffer, C., Sokenou, D.: Improving alignment of crosscutting features with code in product line engineering. J. Object Technol. **6**(9), 417–436 (2007)
Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., Irwin, J.: Aspect-oriented programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 1241, pp. 220–242. Springer, Berlin (1997)
Lee, J., Kang, K.C.: A feature-oriented approach to developing dynamically reconfigurable products in product line engineering. In: Proceedings of the International Software Product Line Conference (SPLC), pp. 131–140. IEEE Computer Society, Washington (2006)
Lieberherr, K.J., Lorenz, D., Ovlinger, J.: Aspectual collaborations—combining modules and aspects. Comput. J. **46**(5), 542–565 (2003)
Liebermann, H.: Using prototypical objects to implement shared behavior in object-oriented systems. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 214–223. ACM Press, New York (1986)
Lippman, S.B.: Inside the C++ Object Model. Addison-Wesley, Reading (1996)
Lopez-Herrejon, R., Batory, D., Lengauer, C.: A disciplined approach to aspect composition. In: Proceedings of the International Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM), pp. 68–77. ACM Press, New York (2006)
McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-age components for old-fashioned Java. In: Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pp. 211–222. ACM Press, New York (2001)
Mezini, M., Seiter, L., Lieberherr, K.: Component Integration with Pluggable Composite Adapters. Kluwer, Dordrecht (2000)
Nicoara, A., Alonso, G., Roscoe, T.: Controlled, systematic, and efficient code replacement for running java programs. SIGOPS Oper. Syst. Rev. **42**(4), 233–246 (2008)
Ostermann, K.: Dynamically composable collaborations with delegation layers. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 2374, pp. 89–110. Springer, Berlin (2002)
Prehofer, C.: Feature-oriented programming: a fresh look at objects. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Lecture Notes in Computer Science, vol. 1241, pp. 419–443. Springer, Berlin (1997)
pure-systems GmbH: Technical White Paper: Variant Management with pure::variants (2004). http://www.pure-systems.com
Rosenmüller, M., Siegmund, N.: Automating the configuration of multi software product lines. In: Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS), pp. 123–130 (2010)
Rosenmüller, M., Apel, S., Leich, T., Saake, G.: Tailor-made data management for embedded systems: a case study on Berkeley DB. Data Knowl. Eng. **68**(12), 1493–1512 (2009)
Rosenmüller, M., Siegmund, N., Apel, S., Saake, G.: Code generation to support static and dynamic composition of software product lines. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 3–12. ACM Press, New York (2008)
Smaragdakis, Y., Batory, D.: Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol. **11**(2), 215–255 (2002)
van Gurp, J., Bosch, J., Svahnberg, M.: On the notion of variability in software product lines. In: Proceedings of the Working Conference on Software Architecture (WICSA), pp. 45–55. IEEE Computer Society, Washington (2001)

Villazón, A., Binder, W., Ansaloni, D., Moret, P.: HotWave: creating adaptive tools with dynamic aspect-oriented programming in Java. In: Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE), pp. 95–98. ACM Press, New York (2009)

Zdun, U.: Some patterns of component and language integration. In: Proceedings of the European Conference on Pattern Languages of Programs (EuroPlop), pp. 1–26. UVK Verlagsgesellschaft mbH, Konstanz (2004)

Zdun, U., Strembeck, M., Neumann, G.: Object-based and class-based composition of transitive mixins. Inf. Softw. Technol. **49**(8), 871–891 (2007)