Technical Report

Nr.: FIN-013-2009

Combining Static and Dynamic Feature Binding in Software Product Lines

M. Rosenmüller, N. Siegmund, G. Saake, S. Apel

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Nr.: FIN-013-2009

Combining Static and Dynamic Feature Binding in Software Product Lines

M. Rosenmüller, N. Siegmund, G. Saake, S. Apel

Arbeitsgruppe Datenbanken

Fakultät für Informatik
Otto-von-Guericke-Universität Magdeburg

Impressum (§ 5 TMG):

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

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

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

Auflage: 52

Redaktionsschluss: 10.09.2009

Herstellung: Dezernat Allgemeine Angelegenheiten,
Sachgebiet Reproduktion

Bezug: Universitätsbibliothek/Hochschulschriften- und
Tauschstelle

Combining Static and Dynamic Feature Binding in Software Product Lines

Marko Rosenmüller, Norbert Siegmund, Gunter Saake
School of Computer Science
University of Magdeburg, Germany
{rosenmue,nsiegmun,saake}@ovgu.de

Sven Apel
Dept. of Informatics and Mathematics
University of Passau, Germany
apel@uni-passau.de

ABSTRACT

Software product lines (*SPL*) are used to build similar programs from a single code base. Programs of an SPL can be distinguished in terms of *features*, which represent units of program functionality that satisfy requirements. Features of an SPL can be *bound* either *statically* at program compile time or *dynamically* at runtime. Both binding times have advantages and disadvantages, as we will explain. However, contemporary techniques and tools for implementing SPLs do not allow a programmer to flexibly choose the binding time per feature. We present an approach that integrates static and dynamic feature binding. It allows a programmer to implement an SPL once and to decide later which features are statically bound and which dynamically. We provide a compiler and report from experiences of applying our approach to two non-trivial product lines. We analyze resource consumption of the SPLs and provide a guideline for optimizing resource consumption.

1. INTRODUCTION

Software product line (SPL) engineering has been applied successfully to many domains to generate tailor-made programs.[1] An SPL is a family of similar programs that can be 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 [2]. Programs of an SPL are generated by composing modules that implement features. Depending on the underlying composition mechanism, features are either *bound statically* (e.g., at compilation time or in a preprocessing step) or *dynamically* (e.g., when loading a program or at runtime). Both binding times have benefits: static binding facilitates customizability without any cost at runtime whereas dynamic binding allows a programmer to flexibly select and bind features at runtime, however, at the cost of performance and memory consumption [1, 12]. We argue that this tradeoff between static customizability and flexibility due to dynamic binding has to be considered in SPL engineering.

A well known example for static composition are preprocessors (e.g., the C/C++ preprocessor), which enable fine-grained customizability and code optimizations. When composing features statically, however, often only a subset of the features is used at the same time and some features might not be used at all. The reason is that we often cannot decide before deployment or runtime whether a feature is needed or not. For example, the required functionality of a *database management system (DBMS)*, deployed on a smartphone or a PDA, depends on the requirements of the applications that use the DBMS which may change over time. A Web browser that stores encrypted passwords in a database requires a DBMS with a data encryption feature. This feature is needed only when the Web browser reads or writes passwords, which is typically not the case most of the time. Since power supply, available working memory, and computing power are limited on mobile devices, such a *functional overhead* is not acceptable.

Dynamic binding helps avoiding this overhead by loading features only when they are needed. Additionally, dynamic binding allows others to independently develop and deploy alternative implementations of features or program extensions, e.g., by using plugins. Dynamic binding even provides means for loading functionality on demand from a network. On the other hand, it increases memory consumption and degrades performance especially when many small extensions are used [12]. This *compositional overhead* can be avoided using static binding if the required features are known before deployment. For example, adapters to the underlying operation system do not need to be bound dynamically.

In previous work, we have shown that we can decide after after development of an SPL whether static or dynamic binding should be used [27]. However, all features have to be bound either statically or dynamically and it is not possible to use a different binding time for each feature. Hence, we have to choose between customizability at runtime and resource optimizations due to static composition. In this paper, we present an approach that integrates static and dynamic binding seamlessly. In contrast to other approaches, we can choose the binding time of an SPL *per feature* after development and generate *dynamic binding units*, which consist of a user defined set of features. Dynamic binding units are composed at runtime depending on the environment and requirements of the running application. Due to static composition of the *inner* features of a binding unit, we achieve fine-grained customizability and performance optimizations. At the same time, the approach provides high flexibility due to dynamic composition of binding units.

The contributions of this paper are (i) an approach for integrating static and dynamic feature binding in SPLs that allows to flexibly switch the binding time per feature, (ii) an evaluation of the approach regarding customizability and resource consumption, and (iii) a guideline for building binding units to optimize resource consumption of an SPL. Us-

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

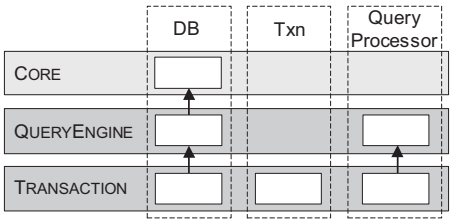

Figure 1: Decomposition of classes (vertical bars) along the features (horizontal bars) in a DBMS.

```
1  //Core implementation
2  class DB {
3    bool Put(Key& key, Value& val) { ... }
4  };
```

```
5  //feature QueryEngine
6  refines class DB {
7    QueryProcessor queryProc;
8    bool ProcessQuery(String& query) {
9      return queryProc.Execute(String& query);
10   }
11 };
```

```
12 //feature Transaction
13 refines class DB {
14   Txn* BeginTransaction() { ... }
15   bool Put(Key& key, Value& val) {
16     ... //transaction specific code
17     return super::Put(key,val);
18   };
19 };
```

Figure 2: FeatureC++ source code of class DB.

ing code transformations and a generated infrastructure for automating dynamic composition, features can be implemented with the same mechanism independent of their binding time. This simplifies development and reduces implementation effort compared to existing approaches. In our evaluation, we present experiences and insights from applying the approach to two non-trivial product lines in order to analyze the impact of dynamic binding on resource consumption. Based on the results, we analyze which features should be composed into a dynamic binding unit. Moreover, we show how resource consumption of an SPL can be optimized by changing their binding units.

2. FEATURE-ORIENTED PROGRAMMING

In this Section, we introduce feature-oriented programming (FOP), a paradigm for implementing SPLs [24, 8] which we use as the basis for our approach. FOP treats the features of an SPL as fundamental elements of the development process. It allows programmers to implement features as increments in functionality [8]. A user creates a concrete program from an SPL by selecting a set of features that satisfy her requirements. The corresponding *feature modules*, i.e., the implementation units of features, are composed to generate a tailor-made program.

In FOP, a feature module consists of classes and class fragments as shown in Figure 1 for a DBMS product line. The DBMS consists of a CORE implementation and two features QUERYENGINE and TRANSACTION, displayed as vertical bars. Class DB provides the interface of the DBMS and classes Txn and QueryProcessor are used to implement transactions and query processing. The two features cut across the implementation of multiple classes shown as white boxes. These class *refinements* implement extensions of a class needed for a particular feature. For example, the basic implementation of class DB is provided in the CORE module and extended in features QUERYENGINE and TRANSACTION (depicted with arrows).

We implemented our approach for combining static and dynamic feature binding using *FeatureC++*,[2] an FOP extension for the C++ programming language [4]. In Figure 2, we depict an excerpt of the FeatureC++ source code of class DB (cf. Fig. 1). Method Put is used to store data provided as key-value pairs. Feature QUERYENGINE adds a new field queryProc and a new method ProcessQuery for processing SQL queries. Feature TRANSACTION adds a new method and *refines* method Put (Line 15). Transaction specific code is added to the beginning of Put (Line 16) and is executed before invoking the refined method using the keyword super (Line 17).

3. STATIC AND DYNAMIC FEATURE BINDING

Based on a feature-oriented DBMS implementation as shown in Figure 1, we can generate different DBMS variants by composing a varying set of feature modules. For example, we can derive a simple DBMS only consisting of the CORE implementation or variants that include the features QUERYENGINE and/or TRANSACTION by combining the according modules. The composition of feature modules can either be done statically or dynamically.

Static composition means to combine the code of multiple features into one executable program and dynamic composition means to apply them in a running program or at load-time. There are different possibilities to categorize the binding time of features in SPLs [13]. In this paper, we refer to static binding if a feature is bound in an application before load-time, e.g., at compilation time, and dynamic binding if it is applied at load-time or after loading an application. In prior work on FeatureC++, we have shown that features can be composed statically or dynamically, using the same code base [27]. In the following, we introduce the code transformations used in FeatureC++ to support different binding times. Nevertheless, the presented concepts are language independent and can be applied to other programming languages as well. A more detailed overview of FeatureC++ can be found in [4, 27].

3.1 Static Binding

In order to support static feature binding, the classes of an SPL have to be composed according to the features selected in the configuration process. Since FeatureC++ is based on a source-to-source transformation to C++, the entire code of the base implementation of a FeatureC++ class and their refinements of all selected features is composed into one compound C++ class. This 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.

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

```
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 };
```

Figure 3: Generated C++ source code of class DB using static binding of Core functionality and feature Transaction.

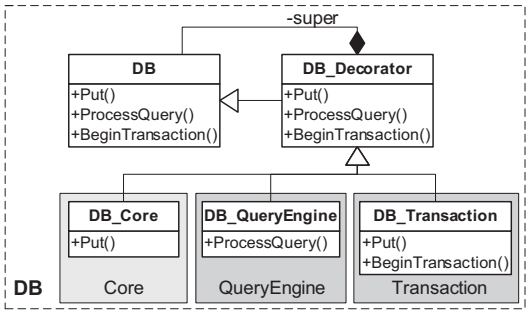

Figure 4: Class diagram of the generated decorator hierarchy for dynamic binding of class DB using features QueryEngine and Transaction.

In Figure 3, we depict the generated C++ code that corresponds to the FeatureC++ code of class DB in Figure 2. This generated code is shown only for illustration and 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 except the code of feature QUERYENGINE are composed into one C++ class. The base implementation of method Put (feature CORE) was renamed to Put_Core (Line 2) to provide a unique name for every transformed method. It is called from its refinement in Line 8. Using this kind of transformation, a C++ compiler can easily inline method refinements since they are composed into the same file. 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++ provides the same performance as code that does not provide such fine-grained customizability [25].

3.2 Dynamic Binding

In order to support dynamic binding of features from the same source code, the classes of an application have to be modified dynamically according to the active features. For example, class DB (cf. Fig. 2) has to be extended dynamically with code of feature TRANSACTION when activating the transaction management of the DBMS. For that reason, we extended the FeatureC++ code generation process to transform the refinement chain of a class into a delegation hierarchy [27]. Similar to the *Delegation Layers* approach [23], we use the *decorator pattern* [14] to compose classes dynamically. Each class thus consists of a decorator for each refinement and a class is combined dynamically by composing the decorators.

For illustration, we depict the class diagram of the transformed class DB in Figure 4. The class is composed from its refinements, which have been transformed into decorators (DB_Core, DB_QueryEngine, DB_Transaction), each belonging to a separate feature. The generated decorator interface (class DB) is used to reference dynamically composed classes within the transformed code and also from external source code. The abstract decorator class DB_Decorator maintains a reference to the predecessor refinement (super reference) and forwards operations that are not implemented by a concrete decorator. The implementation of methods and method refinements are provided by the concrete decorators. For example, method Put (Line 3 in Figure 2) and its refinement in feature TRANSACTION (Line 15) are transformed into methods of concrete decorators DB_Core and DB_Transaction (cf. Fig. 4). Method refinements invoke refined methods by using the super reference of the decorator class.

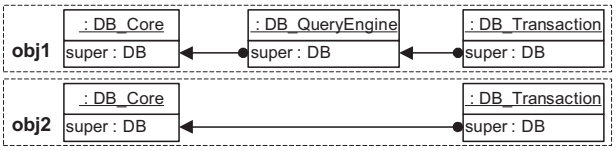

Figure 5: Object diagrams of instances of class DB for two different feature selections.

Feature Classes. When dynamically creating an SPL instance, we have to compose the selected features. We support this *feature instantiation* by using classes to represent features. These *feature classes* are generated in the code transformation process. Much like ordinary classes and refinements, the *feature classes* are also combined using the decorator pattern. For example, when composing feature modules CORE, QUERYENGINE, and TRANSACTION, as shown in Figure 4, there is a feature decorator generated for each feature, which inherits from an abstract decorator, that represents an arbitrary feature of the product line. Each instance of a feature decorator maintains a super reference to the predecessor feature in a composed program.

Class Instantiation. Instantiation of dynamically composed classes means to combine objects of the generated concrete decorator classes according to the selected features as depicted in Figure 5. Shown are two different instances of class DB using the CORE implementation as well as features QUERYENGINE and TRANSACTION. Each instantiated refinement contains a super reference that points to the next refinement in the chain. The dynamically composed objects can be used in the same way as an instance of a regular class and can be modified at runtime by adding or removing instances of decorators. The refinement chain thus corresponds to a linked list of class fragments. Changing the configuration of a class corresponds to insertion, exchange, and deletion of elements of this *refinement list*.

For class instantiation, the feature decorators provide factory methods to create instances of ordinary SPL classes which means creating an instance for each decorator. For example, a generated method newDB() is used to create an instance for each decorator of class DB for a specific SPL instance. Class instances are composed from their decorators within the factory methods of the corresponding features.

Creating an instance of a feature means to simultaneously apply decorators to all classes the feature refines.

Summary. The presented approach provides a single extension mechanism, i.e., class and method refinements, which is independent of the binding time. Hence, a programmer only has to learn and use a single implementation mechanism and does not have to provide a different implementation for each binding time. However, the presented approach only allows to choose between static and dynamic binding for a whole SPL and not for single features.

This results in a *functional* or *compositional overhead* depending on the chosen binding time for the product line. Static binding results in a *functional overhead* when features are included in a program variant but are not used. Dynamic binding, on the other hand, provides more flexibility by dynamically loading features when required. It thus avoids a functional overhead and allows to change or extend a program after deployment. This is achieved by generated code (e.g., decorators) and late method binding, which introduces a *compositional overhead* in terms of memory consumption and performance.

4. COMBINING STATIC AND DYNAMIC BINDING

In order to overcome the limitations of static and dynamic binding, we combine the code transformations presented above to allow a programmer to decide for each feature of an SPL if it has to be bound statically or dynamically.

4.1 Dynamic Binding Units

A feature is typically used in combination with other features and thus dynamic binding of single features is usually not needed but increases resource consumption. An example for a DBMS consisting of five feature modules, is shown on the left side in Figure 6. Usually the transaction management requires feature Logging, so both features should be combined into a single unit for dynamic composition.

This results in groups of feature modules that are statically composed into a single larger *dynamic binding unit*. Lee et al. suggest to define dynamic binding units by grouping features when planning an SPL and to manually implement components according to this decision [19]. We automate this process and generate dynamic binding units as needed at deployment time by composing multiple features statically.

Generating Binding Units. Two examples for generating dynamic binding units are depicted in Figure 6. DB' and DB'' are two transformed product lines (i.e., not concrete products) that support dynamic composition for different application scenarios. For DB', feature modules Core and B-Tree (an index structure for efficient data retrieval) are combined into a single binding unit Base. This binding unit is already a working DBMS. We can compose the features of Base statically because usually a DBMS requires some index structure and we can often decide before deployment which index structures should be used. Similarly, Transaction and Logging are composed into a distinct binding unit Txn. Depending on the application scenario, feature QueryEngine is composed into another distinct binding unit QE in DB' or into binding unit Base in DB'' when

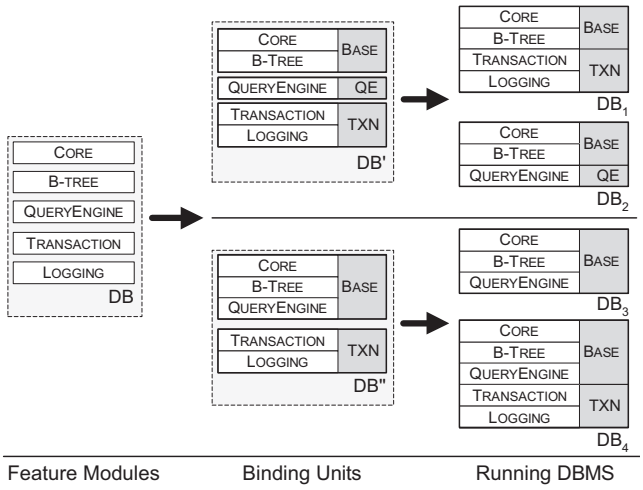

Figure 6: Two possible static transformations of a DBMS product line ($DB \rightarrow DB', DB''$) resulting in dynamic binding units (Base, QE, TXN) and examples of running DBMS (DB_1–DB_4).

query processing is always required. Based on DB' and DB'' we can create a number of different DBMS variants (examples DB_1–DB_4 in Figure 6) by dynamically composing the binding units according to a given configuration, i.e., a list of binding units. Comparing variants DB_2 and DB_3, we see that both provide the same functionality but feature QueryEngine is bound dynamically in DB_2 and statically in DB_3.

Feature composition can be formalized by treating features as functions that modify other features or a base program [8, 5]. The resulting compound module is the source for a next composition step. In our case, a dynamic binding unit itself is a compound feature module and is bound in a dynamic composition process. Hence, dynamic binding often does not mean binding of single features but dynamic binding of larger statically composed feature modules.

We can denote composition of features with $\bullet$ and describe composition of program DB_1 as:

$$Base = BTree \bullet Core \quad (1)$$

$$TXN = Logging \bullet Transaction \quad (2)$$

$$DB_1 = TXN \bullet Base \quad (3)$$

$$= (Logging \bullet Transaction) \bullet (BTree \bullet Core) \quad (4)$$

Equations (1) and (2) represent static compositions resulting in dynamic binding units $Base$ and TXN. Equation (3) represents dynamic composition of these binding units. Combining static and dynamic composition in that way can be problematic if the order of composition matters. The reason is that composition of feature modules is not necessarily commutative [5]. For example, if feature Transaction extends methods of feature Btree, the execution order of the method extensions might be important. When changing the two dynamic units from Equations (1) and (2) to $Base = Transaction \bullet Core$ and $Log = Logging \bullet Btree$, dynamic composition results in the composed program

$$DB = (Logging \bullet Btree) \bullet (Transaction \bullet Core) \quad (5)$$

which is invalid if $Btree$ and $Transaction$ are not commutative. This has to be considered when mixing static and

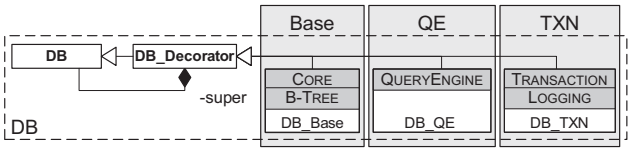

Figure 7: Combined static and dynamic composition of class DB. Generated classes shown as white boxes within light-gray binding units. Transformed code shown as gray boxes within classes.

dynamic composition, e.g., using special code transformations, as we explain in Section 4.3.

4.2 Product Derivation

The process of creating a program from an SPL, i.e., the *product derivation*, can be divided into three steps, (i) configuration, (ii) static transformation, and (iii) dynamic composition. In the first step, a user selects the potentially required features and assigns a binding unit to each feature. In the simplest case, there is only one binding unit.

The subsequent static transformation process generates dynamic binding units and code for automating dynamic composition of these units. Features not selected in the configuration process are not included in any binding unit, which reduces their binary size. There are two extremes: (1) a single binding unit contains all selected features resulting in a pure statically composed program and (2) one binding unit per feature resulting in pure dynamic composition as described above (see Sec. 3). Between these extremes we now support all combinations of static and dynamic binding using static composition of dynamic binding units.

Dynamic binding units are stored in the binary of an application or in extension libraries.[3] FeatureC++ supports dynamic composition of binding units by a composition infrastructure which is generated in the static transformation process. Hence, a programmer does not have to write code for loading and composing dynamic binding units. See [27] for details about dynamic composition, verification of a composition at runtime, etc.

4.3 Source Code Transformations

The code transformations involved in the combined approach are a mixture of static and dynamic composition as already explained. Hence, composing a class from multiple refinements means to first combine refinements from features of the same binding unit and then create decorators for each binding unit. An example corresponding to the binding units of DB' in Figure 6 is shown in Figure 7. Similar to pure dynamic composition, class DB consists of an interface, an abstract decorator, and three concrete decorators (cf. Fig. 4). Now, the concrete decorators themselves are already statically composed classes. Code of the DBMS core and the refinement of feature B-TREE are merged into one class DB_Base that is part of binding unit BASE (cf. Fig. 6). The refinement of feature QUERYENGINE is transformed into class DB_QE, which is part of binding unit QE. DB_TXN is also a statically composed class and part of binding unit TXN. All generated classes are combined using decorators. The needed code transformation is thus a static composition followed by a transformation needed for dynamic composition.

[3]Currently, we support only Windows DLLs.

```
1  //Core implementation
2  class DB {
3    bool Put(Key& key, Value& val) { ... }
4  };
```

```
5  //feature Logging
6  refines class DB {
7    bool Put(Key& key, Value& val) {
8      ... //logging specific code
9      return super::Put(key,val);
10   };
11 };
```

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

Figure 8: FeatureC++ source code of class DB with method Put that is extended in two features.

Storing SPL Context. Class instantiation in a dynamically composed program requires to create an object that corresponds to the configuration of an SPL instance. In FeatureC++, we use objects of dynamically composed *feature classes* (cf. Sec. 3.2) to represent SPL instances. For example, instance DB_2 that includes the binding units BASE and QE of DB' in Figure 6 is dynamically composed from instances of two feature classes, one for each binding unit. When creating an object of class DB as shown in Figure 7, we need to know which SPL instance has to be used. For that reason, we store SPL references within objects. When a class is created the reference to the SPL instance is used for selecting the required concrete decorators. For example, when creating an instance of class DB the SPL instance DB_2 defines the binding units (BASE and QE) 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, if class Txn (cf. Fig. 1) is only part of one binding unit that is statically composed from features TRANSACTION and LOGGING (cf. Fig. 6) we do not need an SPL instance for creating objects of that class because it is independent of the dynamic feature selection.

Nevertheless, when combining static and dynamic composition also an object of a purely statically composed class needs to store a reference to the corresponding SPL instance if it (directly or indirectly) creates instances of other dynamically composed classes. For example, also a class QueryEngine, which is composed statically because it is only part of binding unit QE, has to store a reference to its SPL instance if it creates objects of dynamically composed classes.

Commutativity of Method Refinements. When composing classes, it has to be ensured that the executiuon order of method refinements does not change because they usually have to be executed in a predefined order. Since composition is usually not commutative the execution order would change if static and dynamic composition is mixed. An example is shown in Figure 8. Method Put of class DB

```
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 };
```

```
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 };
```

Figure 9: Generated C++ code of class `DB` with a hook for method refinement.

is extended by features Logging and Transaction. Both method extensions have to be executed bottom-up: first the transaction code has to be executed (Line 15) and afterward the logging code (Line 8). When statically composing the Core implementation and feature Transaction into a single binding unit and feature Logging into a different binding unit, it results in an invalid program when we dynamically compose the resulting binding units. The reason is that when we dynamically add feature Logging, the code of method `Put` defined in Line 7 would be executed first and the refinement in feature Transaction afterwards.

Commutativity of refinements can be provided by a code transformation that ensures the correct execution order of method extensions [3]. For example, we can generate a hook method `Put_hook` as shown in the generated code in Figure 9. The hook is called in Line 10 instead of method `Put_Core`. It is overridden by feature Logging to execute the logging specific code before executing the extended method (Line 16). If feature logging is not present, the hook simply calls the refined method, i.e., `Put_Core` (Line 5).

Summary. When combining static and dynamic composition, the code transformations of a pure static approach are used for static composition of binding units. The binding units are composed using delegation as in a pure dynamic approach. Due to the combination of static and dynamic binding, a dynamically bound feature is often statically composed with other features. Dynamic binding thus not necessarily results in generating decorators for the classes of a feature. A dynamically bound feature might even be transformed using only static composition of its classes and class refinements with classes of other features of the same binding unit.

5. EVALUATION

With an evaluation[4] of static and dynamic feature binding in two SPLs, we demonstrate the applicability of our approach. We analyze the impact of different dynamic binding units on resource consumption and illustrate the benefits

[4]For our evaluation, we use an Intel Core 2 system with 2.4 GHz and operating system Windows XP.

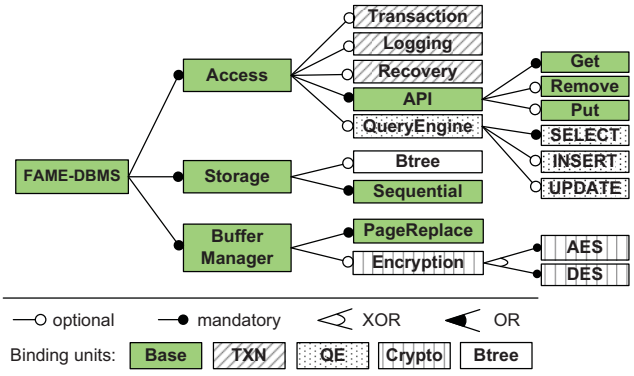

Figure 10: Feature diagram of Fame-Dbms with different binding units.

of combining static and dynamic binding. In Section 6, we discuss the results and analyze when features should be composed statically or dynamically and which features should be combined in a dynamic binding unit.

5.1 Case Studies

Since there are no large SPLs implemented in FeatureC++, we use two product lines for our evaluation, that have been developed in our working group. The first SPL is Fame-Dbms, a DBMS product line for use in resource constrained environments [25]. The second SPL is NanoMail a highly customizable mail client.

FAME-DBMS. Fame-Dbms is an embedded DBMS, i.e., it is embedded into an application and accessed via an API. It was developed for resource constrained device which is possible due to static feature binding. We use it to analyze applicability of the combined approach even though dynamic binding was initially not intended. In Figure 10, we depict an extract of the *feature model* of Fame-Dbms, which is a hierarchical representation of its features, describing optional features and relations between them. We show only features that are relevant for our case study and omit features that are always statically bound like operating system related features. In its current version, Fame-Dbms consists of 56 features with 12400 lines of code.

We compare different variants of Fame-Dbms which use the same configuration including 44 features but different binding units (cf. Fig. 10). This allows us to assess the impact of dynamic binding on resource consumption. Binding unit Base represents a basic DBMS that consists of an API for storing and retrieving data. Binding unit Txn provides transactional access to the database. QueryEngine (QE) is a customizable query engine that supports a subset of SQL by statically composing only the required SQL features [26]. Exemplary, the query engine supports `SELECT-FROM-WHERE` queries. Crypto is a binding unit for data encryption and decryption. Customization of ciphers is done statically by choosing the algorithms, e.g., Aes or Des. Finally, binding unit Btree provides efficient access to data via a B^+tree.

NanoMail. We have developed NanoMail as a highly customizable mail client SPL with 25 features and 6200 LOC. It provides different mail applications from a simple MailNotify application, which only notifies a user if there is unread mail,

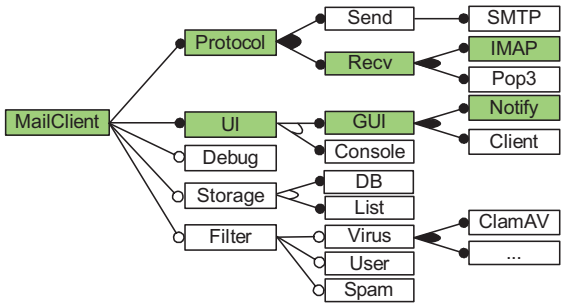

Figure 11: Feature diagram of NanoMail. Feature of a minimal MailNotify application highlighted.

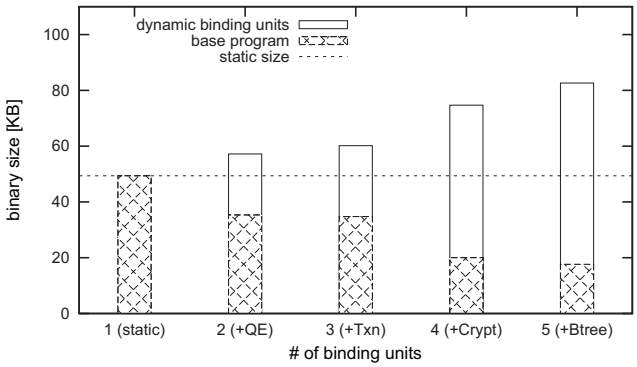

Figure 12: Binary size (base program and dynamic binding units) for variants of FAME-DBMS with an increasing number of binding units.

up to a full mail client with mail storage in a database. The features of NanoMail are shown in Figure 11. Features of the minimal MailNotify application are highlighted as shaded boxes. In our evaluation, we compare variants with equal functionality and varying binding units. For analyzing the impact of fine-grained dynamic customization, we provide several mail filters that are used like plugins. A developer can provide several of those plugins as distinct binding units or group them into one or more binding units.

5.2 Resource Consumption

In order to analyze resource consumption, we measure the binary size, used working memory, and performance of different FAME-DBMS and NanoMail variants and compare their *functional* and *compositional* overhead (cf. Sec. 3). Our aim is to show when static or dynamic binding should be preferred and how dynamic binding units affect resource consumption. In the following, we evaluate both product lines in detail and analyze factors that influence resource consumption.

5.2.1 Binary Size.

The binary size of an application is important when storage is limited (e.g., on a PDA) or when loading binaries in demand. For example, when loading a dynamic binding unit from the network, its size is highly important. In the following analysis, we thus compare the impact of dynamic binding units on binary program size (executable code and static data).

A comparison of the binary size of five variants of FAME-

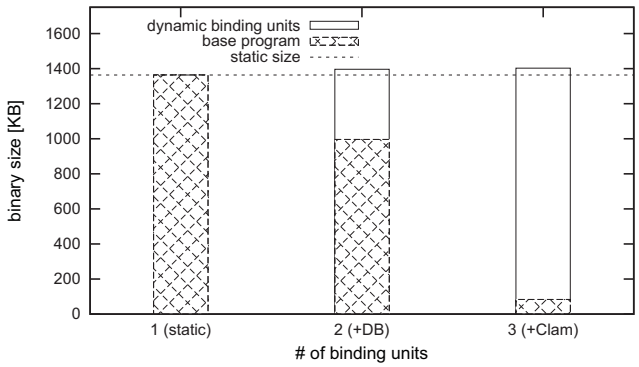

Figure 13: Binary size for variants of NanoMail with varying bining units.

DBMS with equal functionality is shown in Figure 12. 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 (QE, Crypto, TXN, Btree) is extracted from the base implementation and compiled as a distinct dynamically linked library (DLL). The *compositional overhead* of dynamic variants compared to static composition is between 14 % and 40 % (amount above the dashed line in Fig. 12) and increases with an increasing number of binding units. About 80 % of this overhead is caused by generated code (e.g., generated decorators) and the remaining part is code for loading DLLs and a general overhead for DLLs. The generated code includes code for automating dynamic composition (1 KB for loading binding units from DLLs; part of the base application, independent of number of binding units) and generated decorators (13–28 %). However, the *compositional overhead* does not depend on the size of a binding unit, but its tangling with other binding units, i.e., how many other classes are extended and accessed by a binding unit. For example, binding unit QE is the largest binding unit of FAME-DBMS (32 KB) but it causes only half of the overhead of the CRYPTO binding unit. For NanoMail, the relative overhead is only between 1 % and 3 % due to a larger program size (1.33 MB) and well encapsulated binding units. The minimal *compositional overhead* for a binding unit is about 6 KB and grows linearly with an increasing number of binding units. Compared to static binding, this is an overhead between 16 % and 67 % for FAME-DBMS (33 KB–8 KB) and between 0.5 % and 7 % for NanoMail (1.33 MB–84 KB). In detail, the overhead is caused by code duplication, e.g., generated code required in each DLL, and a general overhead for DLLs (5 KB per DLL).

The potential *functional overhead* for FAME-DBMS with respect to the base code of configuration 5, i.e., without loading dynamic binding units, is between 2 KB (3 %; configuration 4) and 32 KB (64 %; configuration 1). In NanoMail, the maximal functional overhead is even larger (up to 94 %) due to the binary size of binding units. For example, the virus filter has a size of 912 KB resulting in a large functional overhead compared to a total binary size of 1.33 MB if the virus filter is not used.

To conclude, our case studies demonstrate that the overhead of binary size highly varies for different binding units and between different SPLs. The compositional overhead is quite high for FAME-DBMS (up to 40 %) and very low for NanoMail (< 4 %). The possible *functional overhead* is for

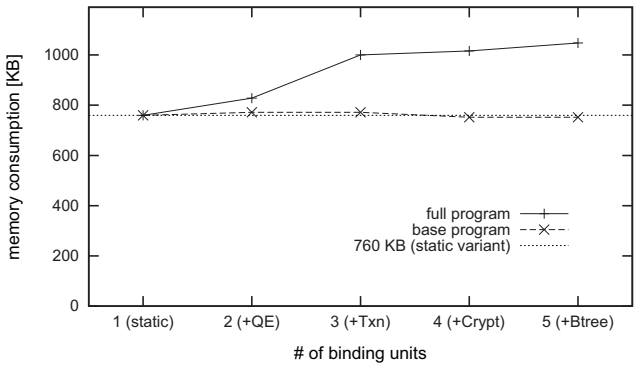

Figure 14: Comparison of consumed working memory of FAME-DBMS variants with an increasing number of binding units. *Full program* variants have loaded the dynamic binding units, *base program* variants have not.

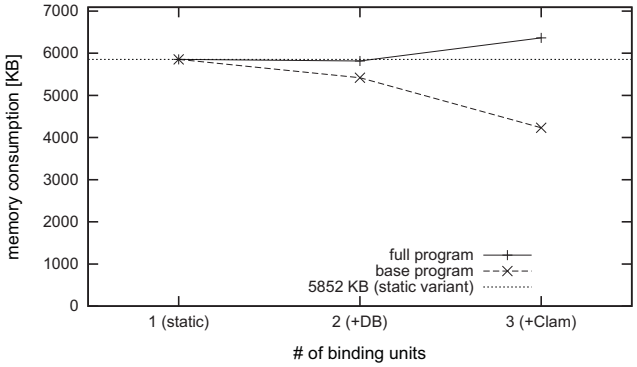

Figure 15: Consumed working memory in NanoMail for variants with different binding units. *Full program* variants have loaded the dynamic binding units, *base program* variants have not.

both SPLs very high (64 %–94 %). Modifying the binding units, both kinds of overhead can be reduced, but the optimal tradeoff has to be found per application scenario.

5.2.2 Memory Consumption.

We show a comparison of consumed working memory for the different variants of FAME-DBMS in Figure 14. Reasons for a varying memory consumption are additional executable code, loaded into working memory, and memory allocated at runtime. Both are analyzed in the following.

Dynamically Allocated Memory. The size of dynamically allocated memory depends on size and number of instantiated objects and additional allocated memory, e.g., for the data buffer in a DBMS. Dynamic binding results in a *compositional overhead* due to an increased size of dynamically composed objects of 12 bytes for each binding unit that extends a class. This is caused by additional references that are stored within objects, e.g., to access the next decorator in a chain of decorators via the `super` reference (cf. Fig. 7). In FAME-DBMS, only a small number of objects is created in the running DBMS which does not result in a measurable overhead. In NanoMail, the size of a mail object in memory increases depending on dynamic customizability by 12 bytes per binding unit that extends the class. However, this is a quite small overhead compared to the size of a usual mail of 2 KB in our case study.

In FAME-DBMS and NanoMail, the allocated memory thus only varies due to a *functional overhead*. It is caused by objects that are not used or by additional fields of a class that are part of an unused binding unit (added via class refinements). For example, the MailNotify application does not retrieve or store mails and thus uses much less memory than a full mail client (1.3 MB instead of 9.8 MB). Hence, if only mail notification is used, there is an overhead of about 8.5 MB. In FAME-DBMS, such an increased memory consumption cannot be observed, because additional features do not allocate a significant amount of memory if not used and the size of the data buffer does not depend on the feature selection.

Executable Program Code. The *compositional overhead* of binary program code also results in an overhead of memory consumption as shown in Figure 14 for FAME-DBMS and in Figure 15 for NanoMail. For dynamic variants of FAME-DBMS with equal features, we observe an overhead between 68 KB and 288 KB (9 %–38 %) compared to the static variant and up to 528 KB (9 %) in NanoMail. The minimal overhead for loading DLLs is about 13 KB per DLL. For small binding units, this means a large overhead compared to the size of the binding unit, e.g., more than 100 % per binding unit for small mail filters in NanoMail as shown in Figure 16. By generating one binding unit for multiple filters, this overhead is avoided. For binding units that consume a larger amount of memory like the virus filter (528 KB) in NanoMail, this is an overhead of about 2 %, which is usually acceptable.

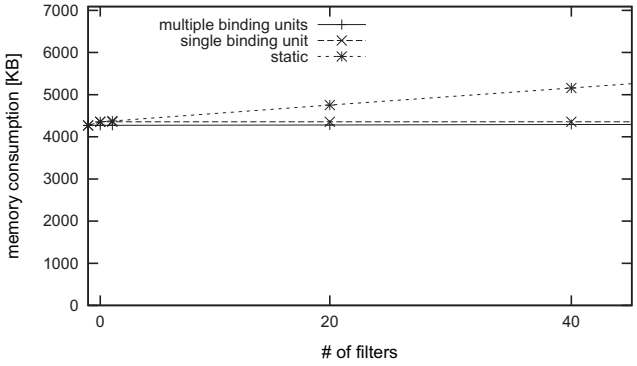

Figure 16: Consumed working memory of NanoMail with an increasing number of mail filters with static composition, dynamic composition with a single binding unit for all filters, and dynamic composition with one binding unit per filter.

In FAME-DBMS, we can observe a slight *functional overhead* of consumed memory due to binary code (i.e., code of unused binding units that are loaded into memory). Dynamic variants of FAME-DBMS use between 752 KB and 772 KB of memory when not loading dynamic binding units, which is an overhead of less than 3 %. For NanoMail, the consumed memory of base programs without binding units varies between 4,232 KB and 5,852 KB (Figure 15). This means a functional overhead in the static variant of 28 % of the total memory if only basic features are used.

To summarize, the *compositional overhead* of allocated

memory increases if a program allocates a large number of small objects that are dynamically composed; and the *functional overhead* of allocated memory highly depends on the application scenario and implementation of features and might be very high. Due to binary program code, a large number or highly crosscutting binding units increase the *compositional overhead* while large binding units affect the *functional overhead* if they are loaded but not used. Overall, the overhead of consumed memory highly depends on size and number of binding units and their implementation but also on user requirements. It thus has to be analyzed for each SPL and application scenario which feature binding and which binding units are the best with respect to memory consumption.

5.2.3 Performance.

Performance of an application is usually specific for a domain. We thus measure the performance of FAME-DBMS using benchmarks for reading and writing data. As shown in Figure 17, the performance decreases with an increasing number of binding units. Comparing dynamic variants to pure static composition, we observe a performance degradation between 5 % (2 binding units) and 28 % (5 binding units). The reason is reduced method inlining and more indirections compared to static variants. For example, 100 % of method refinements can be inlined for static binding. This decreases to about 95 % for configuration 2 and further to 86 % for configuration 5. Additionally, public and protected methods of a dynamically composed class that can be inlined in static variants are also replaced by virtual methods. This is required to access the class from other binding units via its abstract decorator. Private methods and methods of classes that use only static composition are not affected. The compositional overhead mainly depends on the number of method refinements and the invocation frequency of refined methods.

To analyze the startup time of an application, we compare different variants of FAME-DBMS and NanoMail. We observe a general *compositional overhead* for starting a program of about 30 ms per additional binding unit. This is only noticeable for a user if a large number of binding units is used. A *functional overhead* can be observed due to large binding units and complex initialization code. For example, loading the virus filter in NanoMail takes more than 2 s. An influence of unused binding units on performance at runtime is possible if additional code has to be executed, e.g., when scanning mail attachments for viruses if it is not required. However, if the implementation of a feature allows to disable its functionality when loaded, the functional overhead can be avoided. This, however, requires additional implementation effort and is not needed when using dynamic binding as provided with our approach. The functional overhead thus highly depends on the implementation of the features of an SPL.

6. DISCUSSION

In the following, we discuss our results and analyze which features should be composed into a binding unit. Finally, we provide a guideline for building SPLs that support static and dynamic binding.

6.1 Resource Consumption

Extensibility of programs is often achieved with dynamic binding (e.g., in plugins). Static binding, on the other hand, provides fine-grained customizability without any negative influence on resource consumption (e.g., using C/C++ preprocessors). Hence, both binding times are essential for many applications and also for SPL engineering. Our evaluation has shown that *compositional* and *functional* overhead occurs in terms of binary size, memory consumption, and performance when using static and dynamic binding. Other properties not considered here, e.g., consumed network bandwidth, are also affected. For small binding units, the compositional overhead is often high, e.g., for mail filters in NanoMail we observe an overhead of consumed memory of more than 100 % compared to the actual size of a binding unit. This is similar for performance and binary size.

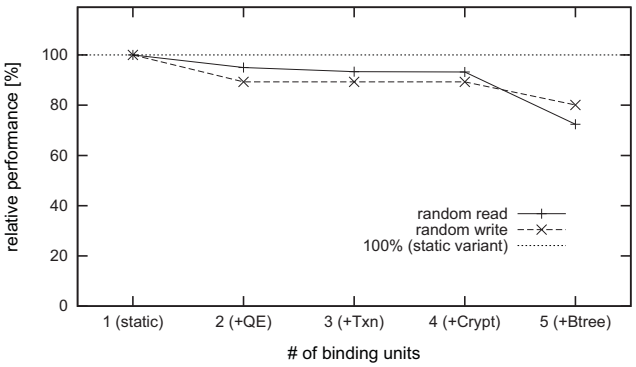

Figure 17: Comparison of benchmarks for reading and writing for different statically and dynamically composed variants of FAME-DBMS. Shown is the relative performance. 100 % is equal to about 3.0 Mio queries / s for reading and 0.8 Mio queries /s for writing.

The *compositional overhead* caused by a binding unit depends on its tangling with other binding units, i.e., on the expected and provided interface. This interface has to support dynamic binding (e.g., using virtual methods) and thus hinders method inlining, introduces indirections, and increases the size of generated code. Hence, it is usually important to aggregate multiple features in larger binding units to reduce the compositional overhead. On the other hand, increasing the number of features per binding unit introduces a potential functional overhead due to unused features.

We summarize our results in Table 1. Shown are the reason and influencing factors for compositional and functional overhead with respect to binary program code, allocated memory, and performance. Memory consumption caused by binary code is not listed separately because it is directly influenced by binary code size. The main difference between compositional and functional overhead is that the compositional overhead is usually caused by a high number of binding units and crosscutting binding units while functional overhead is usually caused by a large size and functionality of binding units. Hence, there is a tradeoff between compositional and functional overhead. Choosing the right features for a binding unit can thus optimize memory consumption and performance.

6.2 Defining Binding Units

The remaining challenge for a domain expert is to find proper binding units to provide the required flexibility while

	Binary Program Code	Allocated Memory	Performance
Compositional overhead			
reason	generated code	dynamic extension of objects	missing inlining, indirections, generated code
influence	number / crosscutting of binding units;	allocation of small extended objects	number of method extensions; invocation frequency
Functional overhead			
reason	code of binding units	allocation in unused features	time for execution of not required functionality
influence	large binding units	memory allocation per binding unit	computational complexity

Table 1: Reasons and main influencing factors for compositional and functional overhead of resource consumption.

minimizing the overall overhead. Because static binding does not exhibit any compositional overhead, it is usually the best choice if dynamic extensibility is not required. Introducing dynamic binding only to reduce the functional overhead might be needed when resources are restricted but not in general. Nevertheless, dynamic binding is mandatory for achieving extensibility after deployment (e.g., for third party extensions) or when the configuration of an SPL has to be changed at runtime.

If dynamic binding is used, binding units should modularize related features to reduce generated code (decorators, etc.) and to enable method inlining between these features. Ideally, the features should not heavily crosscut other binding units. For example, binding unit Crypto (cf. Fig. 10) is well encapsulated and extends only a small number of classes and methods. Most of the classes defined in the binding unit are private to this binding unit, i.e., are not extended by other features and can be composed statically. For finding the optimal binding units, resource consumption of different feature combinations has to be analyzed, which is a complex task. As a simple rule, a large number and a large size of binding units should be avoided since the first increases the compositional overhead and the latter might increase the functional overhead.

To reduce the overhead of a binding unit, different optimizations are possible. *Overlapping binding units*, i.e., binding units that use an overlapping set of features, can be used to shrink the interface of a binding unit or to decrease the number of binding units, which can reduce the compositional overhead. Another optimization is to *split* or *merge* binding units. This is useful, if the requirements change over time and not the whole application should be replaced. For example, binding units that are often or always used can be merged into a single binding unit.

6.3 Conclusion and Recommendations

We have presented an integrated approach for static and dynamic feature binding. To summarize, our approach provides means for:

- changing the binding time of a feature after development using a single implementation mechanism which is independent of the binding time,
- binding only selected features statically or dynamically in order to provide dynamic extensibility and fine-grained static customizability,
- merging dynamically bound features in dynamic binding units to optimize resource consumption.

Since both binding times are used in practice, our approach can replace complex solutions that use a mixture of different approaches which do not allow to change the binding time of a feature after development. With dynamic binding, a developer can achieve dynamic extensibility similar to plugins. With our approach, the same feature can also be bound statically, as it is often done with C/C++ preprocessors. Hence, it unifies approaches for SPL development that are currently used in several domains and even provides customizability for highly resource constrained environments [25]. With respect to the development process, it is much easier to use only one mechanism for extensions (i.e., refinements) and not to switch between different mechanisms (e.g., virtual methods, macros, #ifdef's, etc.). In contrast to C/C++ preprocessors, FeatureC++ furthermore ensures syntactically correct extensions while achieving the same performance [25]. Furthermore, it can simplify development, debugging, and maintenance of an SPL by using static instead of dynamic binding at development time (including static type checking). Finally, a combined approach can also be applied if only static binding is needed and allows for introducing dynamic binding in later versions of an SPL when required.

As shown in Figure 18, our approach provides pure static and pure dynamic composition as supported by many existing solutions but also supports all combinations with varying binding units (shown as triangle). Because there is no optimal size for a binding unit, a domain expert can define binding units per application scenario. As a guideline for configuring SPLs, the following questions have to be answered:

1. Which features have to be bound dynamically?
2. Which dynamically bound features should be composed into one binding unit?

When answering the second question, the compositional overhead can be reduced for a constant number of dynamically bound features by increasing the number of features

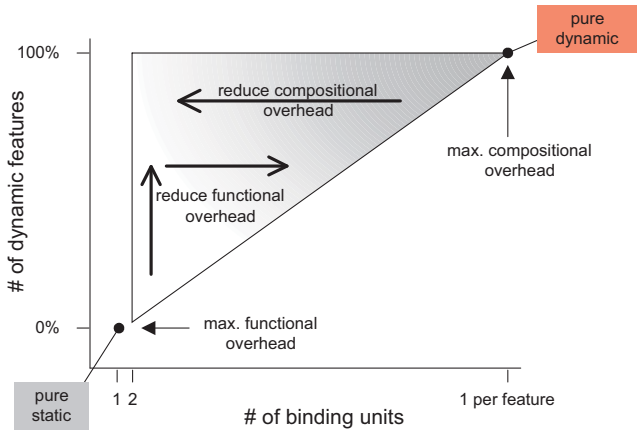

Figure 18: Combining static and dynamic binding to support different extent of dynamic binding and different size of binding units.

per binding unit (arrow in upper part of Fig. 18). The functional overhead can be reduced by increasing the number of dynamic features or by increasing the number of binding units for dynamically bound features (lower left arrows). This allows for optimizations for each application scenario.

Limits of Dynamic Composition. Not all features of an SPL can or should be bound dynamically. For example, it does not make sense to use dynamic binding for mandatory features or features that act as adapters for the operating system. There are also semantic reasons that limit dynamic composition or require a special implementation. For example, the ENCRYPTION feature of FAME-DBMS extends the layout of a database stored in persistent memory with a flag for signaling an encrypted database. Adding or removing such a feature dynamically means that also the data in existing databases has to be changed. Such changes can be highly complex and cannot always be automated. In general, dynamic adaptation is highly complex and there is no general mechanism that allows for exchanging arbitrary features at runtime. Hence, there are special implementations required to support such dynamic modifications that are not in the scope of our work.

Optimizations. The presented approach and its current implementation does not provide an optimized solution. In the following, we describe optimizations that can decrease the compositional overhead of resource consumption due to dynamic binding. Such optimizations cannot avoid the overhead but can reduce it, e.g., depending on the actual application scenario.

SPL Context. As described above, to support dynamic composition an object has to store a reference to the SPL it belongs to. This can be optimized for several use-cases. For example, if there is only a single SPL instance required, we can store this instance in a global variable and avoid additional references within objects.

Memory Allocation and Performance. Currently, code generated by FeatureC++ is not optimized with respect to memory allocation and performance. For example, we could allocate a single block of memory for all concrete decorators when creating an object of a dynamically composed class instead of allocating multiple blocks, one for each decorator.

There are other possible optimizations and there is often a tradeoff between reducing memory consumption and increasing performance which has to be further analyzed.

7. RELATED WORK

There are a number of approaches for software composition that employ different techniques or paradigms to support different binding times. For example, CaesarJ [6] supports static composition based on collaborations and dynamic deployment of aspects. Lee et al. suggest to decide before development which features to implement in one component and to compose the resulting components at runtime [19]. Object Teams support dynamic binding of *teams* which can be used to represent features of an SPL [18]. Composition in Object Teams starts with statically instantiated *activation teams* which in turn activate other teams at runtime. All these approaches require to know the binding time of an implementation unit *before development*. In contrast, our goal is to combine static and dynamic binding based on the same code base. That is, we want to choose the binding time not before deployment to enable reuse of source code even if different binding times are used.

Zdun et al. introduce transitive mixins that generalize composition of classes and objects [29]. In contrast to our approach, transitive mixins are applied to single classes or objects and do not have a representation for features as elements for composition in SPLs. Furthermore, the implementation provided in [29] is build on top of a dynamic approach which cannot provide the benefits expected from static composition.

Other approaches are based on design patterns. Chakravarthy et al. provide with Edicts a solution that supports different binding times using different patterns that are applied to a base program using aspects [11]. Configuration is done by switching the edicts that are manually implemented and not generated. Czarnecki et al. similarly describe how to parameterize the binding time using C++ templates [13]. With a template based program generator this enables automatic configuration of the binding time, e.g., for class extensions. In contrast to our approach, templates as described by Czarnecki et al. do not support simultaneous composition of all classes a feature extends. Both approaches do not allow to compose multiple features statically and compose the resulting binding units dynamically.

There are also approaches that support static and dynamic binding of aspects. AspectC++ supports weaving at runtime and compile time using the same aspects [15]. AspectJ supports weaving advice at compile-time, after compile-time (*post-compile weaving*), and at load-time (when the according class files are loaded into memory).[5] PROSE [22] and Steamloom [9] furthermore support weaving at runtime and may be combined with AspectJ's static weaving. These AOP approaches can be used to support multiple class extensions at the same time like in collaboration based approaches and FOP. However, there is no direct support for composition of whole features according to a feature model. Moreover, it is not possible to compose multiple aspects and bind the resulting module dynamically which would require different binding times per

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

aspect.

Our approach is based on Delegation Layers [23] which supports dynamic composition of features but currently lacks an implementation and does not support static composition. Several other collaboration based approaches and layered designs like Jak [7], Java Layers [10], Jiazzi [21], Mixin Layers [28], Aspectual Collaborations [20], and Context-oriented Programming [17] also support either static or dynamic composition. In contrast to these approaches our solution integrates static and dynamic composition, and aids the developer in dynamically composing SPLs and verification of SPL configurations according to a feature model as described in [27].

Using FeatureC++, dynamic binding units can also be composed at runtime. However, we do not provide a full-fledged solution for runtime adaptation of SPLs. It is not without reason that there is a whole branch of research on runtime adaptation as well as runtime adaptable SPLs [16]. Nevertheless, solutions for runtime adaptation can be built on top of FeatureC++. In this context, our approach integrates traditional SPLs with runtime adaptable SPLs and allows to customize runtime composable binding units statically. Furthermore, the flexible switching between binding times can be used to simplify development of SPLs by temporarily using static binding or removing unneeded features for debugging.

8. SUMMARY AND PERSPECTIVE

We have presented an approach that seamlessly integrates static and dynamic feature binding in SPLs based on the same code base. It allows developers to statically build *dynamic binding units* that are composed at load-time or at runtime of a program. The approach overcomes limitations of pure static and pure dynamic binding and can replace a mixture of static and dynamic approaches with a single implementation mechanism.

In an evaluation, we have analyzed the impact of dynamic binding units on resource consumption. Since there is a tradeoff between *functional overhead* due to static binding and *compositional overhead* due to dynamic binding, the presented approach can be used to enable several optimizations of an SPL by distributing features between different binding units. Such optimizations can be highly complex but tool support could partially automate this process.

In future work, we plan to automate the optimization of binding units, e.g., using static program analysis or monitoring at runtime. As a first step, the FeatureC++ compiler could inform the developer whether a binding unit has a large interface or extends small objects and thus might increase memory consumption or degrade performance.

Acknowledgments

We thank Christian Kästner for comments on drafts of this paper. The work of Marko Rosenmüller is funded by German Research Foundation (DFG), project number SA 465/34-1. Norbert Siegmund is funded by German Ministry of Education and Research (BMBF), project number 01IM08003C. Sven Apel's work is funded partly by German Research Foundation (DFG), project AP 206/2-1.

9. REFERENCES

[1] M. Anastasopoules and C. Gacek. Implementing Product Line Variabilities. In *Proceedings of the Symposium on Software Reusability (SSR)*, pages 109–117. ACM Press, 2001.

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

[3] S. Apel, C. Kästner, and D. Batory. Program Refactoring using Functional Aspects. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 161–170. ACM Press, 2008.

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

[5] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An Algebra for Features and Feature Composition. In *Proceedings of the International Conference on Algebraic Methodology and Software Technology (AMAST)*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer Verlag, 2008.

[6] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development*, 1(1):135–173, 2006.

[7] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 143–153. IEEE Computer Society Press, 1998.

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

[9] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual Machine Support for Dynamic Join Points. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 83–92. ACM, 2004.

[10] R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 285–294. IEEE Computer Society Press, 2001.

[11] V. Chakravarthy, J. Regehr, and E. Eide. Edicts: Implementing Features with Flexible Binding Times. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 108–119. ACM, 2008.

[12] S. Chaudhuri and G. Weikum. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 1–10. Morgan Kaufmann, 2000.

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

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

[15] W. Gilani and O. Spinczyk. Dynamic Aspect Weaver

Family for Family-based Adaptable Systems. In *Proceedings of Net.ObjectDays*, pages 94–109. Gesellschaft für Informatik, 2005.

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

[17] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented Programming. *Journal of Object Technology (JOT)*, 7(3):125–151, 2008.

[18] C. Hundt, K. Mehner, C. Pfeiffer, and D. Sokenou. Improving Alignment of Crosscutting Features with Code in Product Line Engineering. *Journal of Object Technology (JOT) – Special Issue: TOOLS EUROPE 2007*, 6(9):417–436, 2007.

[19] J. Lee and K. C. Kang. A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. IEEE Computer Society Press, 2006.

[20] K. J. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual Collaborations – Combining Modules and Aspects. *The Computer Journal*, 46(5):542–565, 2003.

[21] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-Age Components for Old-Fashioned Java. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 211–222. ACM Press, 2001.

[22] A. Nicoara, G. Alonso, and T. Roscoe. Controlled, Systematic, and Efficient Code Replacement for Running Java Programs. *SIGOPS Operating Systems Review*, 42(4):233–246, 2008.

[23] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2374 of *Lecture Notes in Computer Science*, pages 89–110. Springer Verlag, 2002.

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

[25] M. Rosenmüller, S. Apel, T. Leich, and G. Saake. Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data and Knowledge Engineering (DKE)*, 2009. accepted for publication.

[26] M. Rosenmüller, C. Kästner, N. Siegmund, S. Sunkle, S. Apel, T. Leich, and G. Saake. SQL à la Carte – Toward Tailor-made Data Management. In *13. GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW)*, pages 117–136, 2009.

[27] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. 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)*, pages 3–12. ACM Press, 2008.

[28] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[29] U. Zdun, M. Strembeck, and G. Neumann. Object-based and Class-based Composition of Transitive Mixins. *Information and Software Technology*, 49(8):871–891, 2007.