Supporting Multiple Feature Binding Strategies in NX

Stefan Sobernig Gustaf Neumann Stephan Adelsberger
Institute for Information Systems and New Media
WU Vienna
Austria
{firstname}.{lastname}@wu.ac.at

ABSTRACT

Feature-oriented programming (FOP) environments restrict implementers of software product lines to certain implementation choices. One is left with the choices between, for example, class-level or object-level extensions and between static or dynamic feature bindings. Once developers have chosen an FOP environment, they are confined to its capabilities. This choice is typically made at an early development stage causing an unwanted lock-in. We present a feature-oriented development framework based on dynamic, object-oriented constructs that allows for deferring such design decisions by piggybacking on first-class language entities (metaclasses, mixins) for feature-oriented programming. To demonstrate the feasibility, we implemented a prototypical framework in the scripting language NX. NX provides the required OO infrastructure for dynamic software evolution : a reflective language model, metaclasses, multiple class-based inheritance, decorator mixins, and open entity declarations. We exemplify the approach based on a Graph Product Line.

1. INTRODUCTION

A software product line (SPL) provides a common code base for a family of related software products and a product line model (e.g., a feature model) specifying the set of valid products which can be built from the product line. An important approach to constructing software product lines in an object-oriented (OO) programming environment are collaboration-based designs [23].

In a *collaboration* [1], objects and classes interact by exchanging messages to realize an integrated piece of functionality. The base product is a collaboration implementing a domain model using a mix of OO composition strategies (e.g., a structure of associated objects and classes). The implementation of a feature is a code unit that is designed to extend the collaborations of the base product. The target products (the instances of the SPL) are built from a set of software assets comprising the base product and feature implementations selected by a valid configuration of the product line model. Therefore, the composed products embody the collaboration-based designs of the base product and of the feature modules. This step of building an SPL instance based on *feature composition* is supported by object-level composition techniques as well as by dedicated approaches for feature-oriented programming (FOP ; [23]).

In order to implement a software product line, various design decisions must be made on the base product and on the individual feature implementations. Important decisions are : How is the set of assets organized into separate code units to be combined into a final product ? How can feature cohesion [9] be achieved for these code units ? How should object collaborations and their feature-specific refinements be expressed and made explicit ? Which OO extension mechanisms could/should be used for feature composition ? Is first-class code representation (data structures, objects, classes) of the assets required ? Should the target product be a code unit (such as a class) that should be further reusable and refinable ? What is the desired feature binding time ? In other words, in which program phase are the feature implementations included into the product : at design time, at compile time, at start-up time, or at runtime ? What is the desired/required product granularity : Is the product to be represented as a collaboration of objects or of classes ?

By adopting certain object-compositional techniques or a particular FOP toolkit, many of these decisions must be made comparatively early in an SPL development cycle. For example, the chosen FOP approach (e.g., rbFeatures [6]) and the underlying OO language (Ruby) determines certain decisions due to the programming language's OO model. For Ruby, this model is class-centric with single, class-based inheritance and a form of mixin composition based on dynamic superclass injection. The FOP approach might also provide mandatory abstractions for features (e.g., feature classes) and products (e.g., product classes). Collaborations of classes and objects could be expressed in a language-supported manner (e.g., by declaring a namespace per collaboration). As for the feature composition, the toolkit might adopt a static, class-level approach (e.g., by generating a composed source representation of a class structure at the SPL build time).

These decisions appear to be prematurely taken as they come bundled with the chosen FOP framework and the underlying programming language. After having implemented the product line to a large extent, revisiting any of those decisions at some later time (e.g., due to changed requirements) might even require a complete re-implementation in a more fitting FOP environment.

FOSD'12, September 24–25, 2012, Dresden, Germany.
Copyright 2012 ACM 978-1-4503-1309-4/12/09 ...$15.00.

In this paper, we present an FOP framework based on dynamic OO constructs that allows for deferring the design decisions such as the feature binding time and the product granularity. To demonstrate the feasibility of the approach, we implemented the framework prototypically in the dynamic, object-oriented scripting language NX [11]. This prototype showcases the required OO infrastructure for dynamic software evolution [17], comprising a reflective language model, metaclasses, multiple class-based inheritance, and composable inheritance hierarchies [15]. With this, our approach provides means to defer decisions about . . .

– the representation of feature modules,
– the feature binding time (static, dynamic), and
– the composition granularity (class, object).

The remainder of this paper is structured as follows : In Section 2, we elaborate on our motivation to support variable SPL implementation decisions and we identify a set of challenges. In Section 3, we focus on the necessary language support to address these challenges, and introduce a lightweight realization based on the scripting language NX. Then, we briefly compare our approach with related work on SPL implementation techniques (Section 4). We close with a summary and an outlook (Section 5).

2. VARIABILITY IN FEATURE COMPOSITION DECISIONS

In the following, we will use the common example of a Graph Product Line (GPL) to illustrate our approach. This product line example has been used in closely related work on FOP [20, 16] and will, therefore, facilitate comparing the approaches. As a product line model, the GPL is shown as a feature diagram in Fig. 1. The GPL is modeled as a family of products which implement different types of graphs (colored, weighted, directed, undirected, edge-labeled, etc.), different representation strategies (e.g., adjacency or incidence lists), and support algorithms (e.g., for graph traversals). For this paper, we only look at selected features. The feature `colored` adds coloring support to graph edges. The second feature, `weighted`, adds labeling support to graph edges to store and to attach weightings to edges. As shown in the feature diagram in Fig. 1, these two exemplary features are both optional (depicted by the empty dot markers) and simultaneous. That is, this product line model allows for four valid products : plain graphs, colored graphs, weighted graphs, and, both, colored *and* weighted graphs.

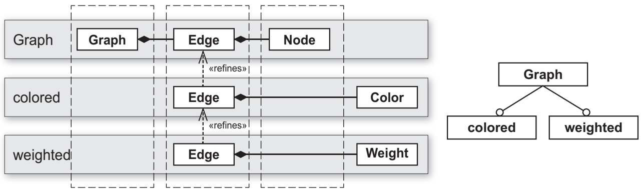


Figure 1: A Collaboration-based Design (left) and a Feature Diagram of the GPL (right)

The design of the exemplary GPL is collaboration-based and layered (see Fig. 1, on the left). The graph base product is implemented by a collaboration of three entities : `Graph`, `Edge`, and `Node`. The two feature implementations of `colored` and `weighted` refine these base entities (i.e., `Edge`) in an incremental manner (e.g., by adding to the printing facility of the edges). For such designs, numerous feature implementation techniques have been proposed, for example : mixin layers [23], delegation layers[16], virtual classes [2], and decorator layers [19].

2.1 Feature Binding and Composition Levels

Including a feature into a program is referred to as *binding* a feature. Feature binding can occur at several binding times, with each programming language and runtime environment providing a characteristic set of binding times (preprocessing time, compile time, load time, program execution time), and in certain binding modes (i.e., fixed, changeable, or dynamic ; [3]). Static feature binding occurs at an early binding time and represents an irrevocable inclusion of a feature into a program. Forms of dynamic feature binding allow for deferring feature inclusion to later binding times (e.g., during program execution) and for revoking the inclusion decision during the lifetime of a program.

Composing feature implementations can be performed at different levels of abstraction : the object, the class, the method, the sub-method, or the statement levels. For the scope of this paper, we limit ourselves to objects and to classes (see Fig. 2), the primary abstraction levels in object-oriented, collaboration-based designs [23]. At the *class level*, the derived product is represented by a single composed class or a composed, collaborative class structure to be instantiated. At the *object level*, the product is embodied by a single composed object or a composed object collaboration.

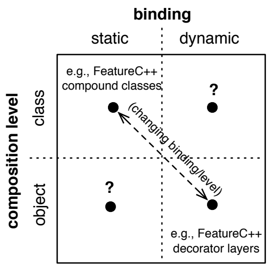


Figure 2: Variable feature composition

Along these two dimensions, FOP approaches [20, 16, 23] fall into four categories (Fig. 2). For example, the code generation approach of Rosenmüller et al. [20] covers static class-level (FeatureC++ compound classes) *and* dynamic object-level feature compositions (FeatureC++ decorator layers). By offering multiple feature binding strategies, an FOP environment realizes *composition variability* :

Changeable feature binding times. This allows the product line implementer to derive products from one code base which can benefit either from static feature binding (e.g., allowing for code-level optimization, avoiding binding overhead in execution times, minimizing the memory footprint, removing otherwise dead code) or from dynamic feature binding (e.g., product reconfiguration during runtime, lazy acquisition of platform-specific product refinements).

Changeable composition levels. Closely related, one might learn that certain feature implementations should only be applied to selected product instantiations (depending on runtime conditions). Class-level implementations of product line assets represent a family of product instantiations. To obtain a handle on one of these instantiations (i.e., an instance), idioms such as singleton classes to represent product instances at the class level must be devised (see, e.g., [2]). Alternatively, an FOP framework might support a transition to an object-level composition.

Mixed binding/composition strategies. Finally, consider the requirement to mix different binding strategies and composition levels. First, a product is composed statically at the class level (e.g., through source code generation), resulting in a class collaboration composed from a base product and certain features. To enact the collaboration, instances of the collaborating classes must be created. This instance structure could be further extended dynamically and at the object (instance) level by another feature during runtime, independently from further instantiations of the product's class collaboration. This requires the flexibility to change the composition strategy at arbitrary times (and not only at SPL build time [19]).

2.2 Challenges

A survey of closely related work [19, 7, 1, 16, 23, 24] reveals important challenges of providing composition variability :

Single code base. At the time of designing and implementing the product line assets (e.g., the core and the feature modules), adopting several level/binding combinations should not require the redundant and diverging implementation of features [19]. Code specific to a given feature, bindable both statically and dynamically (e.g., `weighted` in the GPL), should not be kept in two different implementation variants. Also, the feature implementations should not contain binding-specific boilerplate code (e.g., wrapper code for feature-module loading at runtime). If neglected, there is the risk of introducing code clones [22].

Avoiding decomposition mismatches. In an object-based decomposition of a layered, collaboration-based design, the collaborations (`Graph`, `colored`, and `weighted` in Fig. 1) and the collaboration parts (`Graph.Edge`, `weighted.Edge`) are represented by distinct objects. For the client of a collaboration-based product instantiation (a weighted graph object), the parts of a complex collaboration form single conceptual entities (e.g., the composed, most refined edge kind). This decomposition mismatch can entail a self-problem during method combination and method forwarding [8, 16].

Composition locality. Critical operations (e.g., message sends) on and within a composed collaboration (a weighted graph) should be local to the composed collaboration. The composed collaboration so sets the context for, e.g., constructor calls [24, 16].

Symmetry : Binding and unbinding. For feature bindings to be fully dynamic, the binding operation must be reversible [7]. This is also necessary to form valid products during runtime when facing mutually exclusive features.

Product-bounded quantification. For binding feature implementations, quantification [5] refers to evaluating selection predicates over a program structure (an AST, an interpreter state) to match code units (objects and methods in the base program) for performing transformation, weaving, and intercepting operations on them. Support for dynamic feature binding allows one to create multiple products (and product instantiations) side-by-side [19] ; for example, multiple graph products each with a different feature configuration. This requires the client code to manage multiple feature compositions. Reconfiguring selected products (e.g., unbinding the `colored` feature) must preserve the feature composition of the remaining products through tailorable quantification statements.

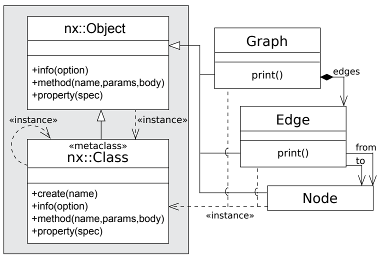
Host language integration. The product code resulting from a feature composition step should be usable directly from native applications written in the host language. As an example, consider the plain C++ support as discussed in [19]. Any unwanted interactions between the FOP infrastructure (e.g., collaborations) and the host language features used to implement them (e.g., classes and class inheritance systems, the type system) must be controlled [23]. For example, if the product derived was represented by a collaboration structure implemented by a set of (nested) classes, these classes would have to remain refinable by means of native subclassing (the inheritance hierarchy) without breaking the collaboration semantics (the extension hierarchy).

3. LANGUAGE SUPPORT FOR VARIABLE FEATURE COMPOSITION

In this section, we present an approach to supporting both static and dynamic, as well as class- and object-level feature composition. The approach adopts established high-level, object-oriented abstractions for object composition (i.e., decorator mixins, class-based multiple inheritance), object/class aggregation, and metaclasses. While applicable to several language environments providing these constructs, we showcase the approach for the dynamic, object-oriented scripting language NX [11] because it provides built-in support for all of these composition operations. Therefore, NX is a convenient test bed for an implementation study.

3.1 The Scripting Language NX

NX is a highly flexible, Tcl-based, object-oriented scripting language. NX is a descendant of XOTcl, a language designed to provide language support for design patterns [13]. The object system of NX is rooted by a single class : `nx::Object`. All objects are instances of this class. In NX, classes are a special kind of object providing methods to their instances and managing their life-cycles. These class objects (simply *classes*, hereafter) are instances of the metaclass `nx::Class`. NX supports object-specific behavior : Objects can carry behavior distinct from the behavior specified by their class. This behavior can be defined in object-specific methods and by decorator mixins (see per-object mixins in [12]). The object system is highly flexible, the relations between objects and classes and among classes can be changed at arbitrary times. NX supports dynamic software evolution [17] by supporting dynamic state and behavior changes at runtime, as well as dynamic changes to program structure and to program composition [10]. In the remainder, we concentrate on the language features of NX relevant for supporting feature-oriented programming. Throughout the section, we refer to the collaboration-based design of a Graph Product Line (GPL, see Fig. 1).



(a) UML

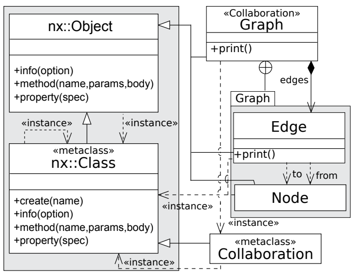
```
nx::Class create Edge {
  :property from
  :property to

  :public method print {} {
    # ...
  }
}
nx::Class create Node
nx::Class create Graph {
  :public method print {} {
    # ...
  }
}
```

(b) NX

Figure 3: NX and GPL Base Classes

Creation of Objects and Classes in NX. Fig. 3a shows the base classes `nx::Object` and `nx::Class` with a subset of their methods (`create` for object construction, `info` for object introspection, `method` and `property` for member declarations). While the instances of `nx::Object` are objects, the instances of `nx::Class` are classes. Since the class of a class is a metaclass, `nx::Class` is a metaclass.

To define the basic GPL class structure, one constructs the corresponding application classes `Graph`, `Edge`, and `Node` using the method `create` of `nx::Class`. Per default, the superclass of the application classes is the root class `nx::Object`. The code snippet in Fig. 3b defines the classes modeled in Fig. 3a. NX provides properties as attributes with generated accessor methods. For the class `Edge`, the two properties `from` and `to` are defined. The classes `Edge` and `Graph` declare two `print` methods. Note that all these artifacts are created at runtime (i.e., upon evaluation of the script) via methods defined by the metaclass `nx::Class` (the methods `create`, `method`, and `property`).



(a) UML

```
 1  nx::Class create Collaboration \
 2      -superclass nx::Class
 3
 4  Collaboration create Graph {
 5    :property name
 6    :property edges:0..n
 7
 8    :public method print {} {
 9      # ...
10    }
11
12    nx::Class create [self]::Node
13    nx::Class create [self]::Edge {
14      :property from
15      :property to
16      :public method print {} {
17        # ...
18      }
19    }
20  }
21
22  Graph create graph1
```

(b) NX

Figure 4: Metaclasses and Object Aggregation

***Object and Class Aggregation in NX*.** In order to group multiple object and class definitions, we use dynamic object aggregations [14]. The aggregation relationship realizes a part-of relationship commonly used in object-oriented designs. NX supports object aggregation via nesting objects as trees based on their names. In NX, objects and classes are explicitly named. The object aggregation in NX is based on object naming similar to file system paths : The name of an aggregated object is prefixed by the name of the parent object, using `::` as a separator. In the same way that objects can contain other objects, classes can contain other classes. This is a consequence of classes being objects.

Later, we will use object aggregation for two purposes : To express which classes interact within a collaboration and to define feature modules. A collaboration is then modeled as a class containing the interacting classes (the collaboration parts) as its child objects. In Fig. 4a, we define the `Collaboration` concept as an NX metaclass (lines 1–2). A metaclass is a specialized `nx::Class`. The metaclass will get more behavior later. Then, the NX class `Graph` is defined as an instance of the `Collaboration` metaclass (lines 4–20). This collaboration contains the child classes `Graph::Edge` and `Graph::Node`. Note that these class names are prefixed by the name of the actual collaboration `Graph`. The collaboration class `Graph` can be used like any ordinary NX class : It can own properties and methods (see lines 5–10 in Fig. 4b), it can be instantiated (see line 22) and subclassed.

In the UML, the collaboration class `Graph` is represented using a UML class stereotyped `«Collaboration»` and an attached, equally named UML package (see Fig. 4a). The containment relation between the collaboration class (`Graph`), the package (`Graph`), and the nested classes (e.g., `Graph::Edge`) is modeled using the cross-hair notation ⊕.

3.2 Variable Feature Composition in NX

Below, we define the code assets of a SPL to be used as the single source for static and dynamic feature binding. The same assets are also used to compose products at the class level and at the object level. Furthermore, we show how to combine these feature composition techniques. Moreover, the implementation techniques honor the previously identified requirements (see Section 2.2). We outline two techniques for dynamic feature binding at the object level and at the class level, respectively. Then, we elaborate on turning dynamically composed product representations into their source representations to be used for static feature binding. Our approach differs from prior approaches in two respects : First, in a dynamic scripting environment as NX, dynamic feature binding is the native mode. Second, we support *all* four combinations of composition levels (object, class) and binding modes (static, dynamic) while existing approaches are mostly limited to two : static class-level and dynamic object-level bindings [19].

3.2.1 Common Assets

In a first step, we create the code assets of the Graph Product Line (GPL) as aggregated objects. The assets consist of the collaboration implementing a basic graph and the feature modules (`weighted` and `colored` ; see Fig. 5). This allows us to address and to handle the assets as objects in our minimal FOP framework. As objects, the product line assets can be easily introspected and modified using common programming idioms.

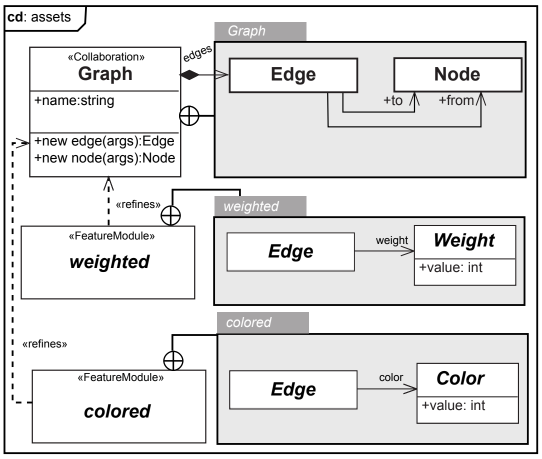


Figure 5: Assets used in the GPL

The collaboration classes (`Graph` in Fig. 5) are both class objects and namespaces. As namespaces, they add namespace qualifiers (`Graph::*`) to disambiguate the objects representing collaboration parts (e.g., `Graph::Edge` vs. `weighted::Edge`). As objects, they provide a collaboration interface to client objects. Most importantly, the collaboration interfaces expose factory methods (`new edge()`, `new node()`) to instantiate refined, collaboration-specific variants of the contained objects. The generated factory method supports composition locality for clients (see Section 2.2).

```
1  nx::Class create FeatureModule -superclass Collaboration {
```

```
2    :property {partial:switch true}
3  }
4
5  FeatureModule create weighted {
6    nx::Class create [self]::Weight {
7      :property {value 0}
8    }
9    nx::Class create [self]::Edge {
10     :property weight:object,type=::weighted::Weight
11   }
12 }
```

Likewise, we define feature module classes as specialized collaborations (see line 1 in the listing above). In Fig. 5, the corresponding UML classes are tagged as «FeatureModule». In contrast to collaboration classes, feature modules are not meant to be instantiated directly. Feature modules represent intermediate and abstract collaborations. They are marked *abstract* in their UML representation in Fig. 5. As a consequence, the previously mentioned factory methods are not generated after having included each feature module, but rather for the composed, final collaboration.

3.2.2 Dynamic Class-Level Feature Binding

For class-level feature composition, the objective is to derive a class structure from the collaboration classes which forms the configured product (`Graph` and `weighted` for a weighted graph). In NX, this class structure can be built on the fly, by generating a metaclass based on the product line assets. In order to build a graph product named `G1` with weighted edge support from the GPL, we need the base collaboration `Graph` and the feature module `weighted` (see Fig. 6). In this scenario, the composition artifact `G1` is

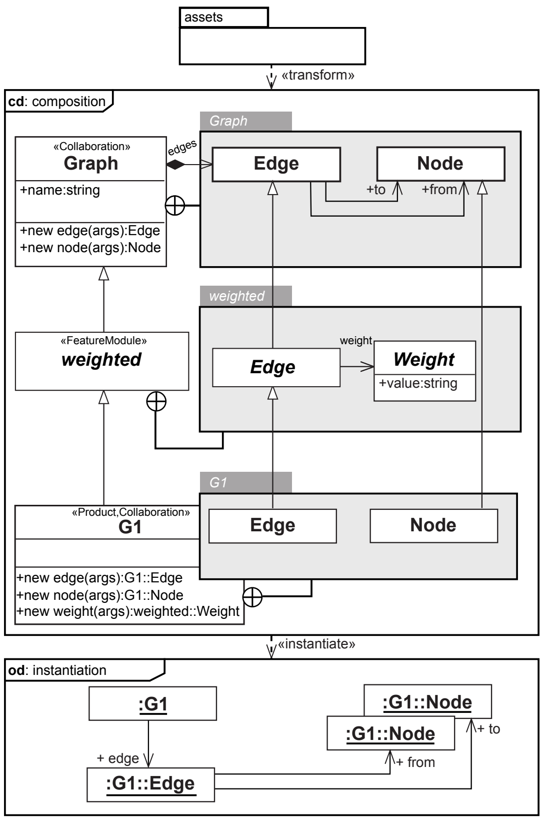


Figure 6: Class-level Feature Binding

again a collaboration class with two nested classes `G1::Edge` and `G1::Node`. This class structure represents the derived «Product» which, like any other class, can be instantiated and subclassed. Since the result of the asset transformation is a freshly configured class, the constructor of its metaclass is the natural place for performing the transformation. The input to this generative step are the base collaboration and the respective feature modules. We add these two properties to the definition of the `Collaboration` metaclass in Fig. 7a, lines 1–2. Upon creating a new class from the metaclass (Fig. 7a, lines 5–7), the constructor of the `Collaboration` metaclass performs the following steps :

1. Compute the collaboration parts based on the base class and the configured feature modules.
2. Compute the extension hierarchy for the collaboration classes and the collaboration parts.
3. Add the collaboration classes of the feature modules as superclasses of the base class.
4. Create additional nested classes in the generated collaboration, one for each collaboration part. Then, the collaboration role classes are combined using multiple inheritance, following the computed extension hierarchy.
5. Add factory methods as instance methods of the generated class for creating instances of the collaboration parts.

The last step provides composition locality (e.g., by returning instances of `G1:Node` rather than `Graph::Node`). The result of composing the base `Graph` and the feature module `weighted` is shown in Fig. 6.

```
1 Collaboration property {base [self]}
2 Collaboration property \
3   {features:0..n ""}
4
5 Collaboration create G1 \
6     -base Graph \
7     -features weighted
```

(a)

```
1 set g [G1 new -name "A G1 instance"]
2 $g print
```

(b)

```
1 G1 public method print {} {next;}
2
3 nx::Class create MyGraph -superclass G1 {
4   :public method print {} {next;}
5 }
```

(c)

Figure 7: The Generated Collaboration Class `G1`

Finally, the resulting class-level product `G1` can be instantiated (see Fig. 7b ; see also the last transformation in Fig. 6). The dispatch upon the `print` method (see line 2) proceeds from `G1` to `weighted` and then `Graph`. By leveraging the built-in NX object and class generation mechanism, client components of the class-level product can use it as an ordinary class. The collaboration class (`G1`) can be refined further either by providing methods to it (line 1 Fig. 7c) or by subclassing (lines 3–5). The same holds for the collaboration parts (`G1::Edge`, `G1::Node`). NX's built-in object system introspection is used during the above transformation steps to query the child objects of the collaboration classes and to extract their object names.

3.2.3 Dynamic Object-Level Feature Binding

In the second dynamic binding scenario, the feature composition is performed at the object level using the common code assets (see Section 3.2.1). At the object level, an instance of a collaboration class and its child instances, representing the collaborating parts, are the binding targets.

As an example, we refer to the collaboration class `Graph` and a feature module `weighted` to form a weighted graph product (see Fig. 9). In an object-level composition, one can specify the feature composition either at the time of object construction (called dynamic feature binding in [20]) or at a later time during the object's life span. Similarly, we can remove feature modules from the created graph at later times.

```
1 Graph create g1 -name "A plain graph"
2 g1 edges add [g1 new edge \
3          -from [g1 new node] \
4          -to [g1 new node]]
5 g1 print
6 # ...
7 Graph property {
        features:0..n,incremental ""}
8 # ...
9 g1 features add weighted
```

(a)

```
1 g1 edges add [g1 new edge \
2          -from [g1 new node] \
3          -to [g1 new node] \
4          -weight [g1 new weight]]
5 g1 print
6 # ...
7 g1 features add colored
8 # ....
9 g1 features delete weighted
```

(b)

Figure 8: The Refinable Graph Instance `g1`

Following Fig. 9, we create an instance of the plain `Graph` collaboration called `g1` (line 1, Fig. 8a). In lines 2–4, a single edge is added to the newly created graph. In line 5, we print the Graph using the method implementation provided by `Graph`. To manage the inclusion and the exclusion of feature modules, we provide a special-purpose property `features` to the family of `Graph` objects (line 7, Fig. 8a). As for introspection, this property can be queried by client objects or from within a collaboration during runtime to retrieve the currently active feature set. As for intercession, the `features` property supports reconfiguration of a given `Graph` instance every time the value of the property `features` changes. NX provides various hooks to capture property changes. When feature modules are added or removed (see lines 7 and 9, respectively, in Fig. 8b), the following steps are performed :

1. Compute the collaboration parts based on the class of the current object and the configured feature modules.
2. Compute the extension hierarchy for the collaboration classes and the collaboration parts.
3. Add the collaboration classes of the feature modules as decorator mixins to the current object.
4. Add factory methods as per-object methods to the refined object. These factory methods are responsible for registering the decorator mixins to newly created instances of the collaboration parts.

The refined graph instantiation `g1` is the product representation of a weighted graph (as identified by the `«Product»` tag in Fig. 9). Since the factory method of `weighted::Edge` is mixed into the object `g1`, `new edge()` calls on object `g1` to accept the additional `weight` argument (line 4, Fig. 8b). The `print` method provided by the `weighted` feature is resolved (line 5, Fig. 8b).

Since NX provides language support for decorator mixins, adding decorators does not require any kind of code refactoring or the generation of intermediate code structures (such as the decorator generator in [20]). The NX decorator mixins preserve the self-context throughout the composed collaboration, thus avoiding issues pertaining to decomposition mismatches (see Section 2.2).

As already stated, the running GPL example only depicts the most basic binding scenario, with a single feature module being included. Also, there are no class inheritance relations between the collaboration parts to be preserved by the extension hierarchy. However, NX supports the construction of complex decorator mixin chains and the decorator mixins can form their own inheritance structures to allow for incremental mixin implementations [25]. As a result, multiple feature modules (and the underlying `«mixin»` relations) can be added and deleted (lines 7 and 9, Fig. 8b) to support feature binding *and* feature unbinding (see Section 2.2).

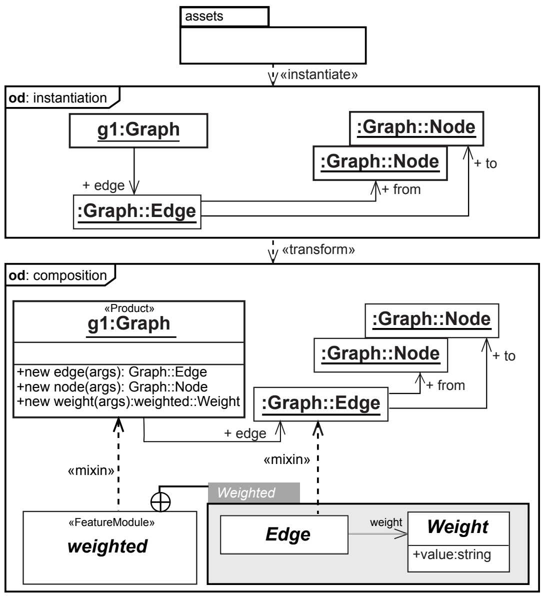


Figure 9: Object-level Feature Binding

3.2.4 *Static Feature Binding*

Under static feature binding, feature implementations are included into an application before load time, typically by a source-code generator or a specialized compiler frontend. This definition targets especially at languages which provide binding times prior to the actual runtime (e.g., compile time). Transferring the notion to a dynamic languages reveals two properties of static binding (see also Section 2.1) : (a) Generating a tailored source code representation of a valid product (e.g., of the readily composed `G1`) and (b) disallowing product reconfigurations (i.e., the product code structure is fixed). The latter property is commonly motivated by baking code-level optimization (for a particular resource constrained platform) into a product and by avoiding the time penalties of dynamic feature binding [19].

Dynamic and reflective languages such as NX can meet property (a) by serializing [18] a given product of the SPL. NX provides a flexible serializer infrastructure capable of streaming objects and classes into source code, reflecting their current configuration state. Therefore, we can serialize the object-level and class-level products with no effort :

```
1 package require nx::serializer
2
3 foreach fm [FeatureModule info instances] {
4   puts [$fm serialize]
5 }
6 puts [G1 serialize]
7 puts [g1 serialize]
```

The above snippet showcases the loading of the NX serializer and its instrumentation to create a script from the SPL instances as specified in the previous two sections. Serialization is supported both for the class-level and for the object-level compositions (see lines 6 and 7 above).

From property (b) it follows that the composed product with its refinement relations must not be changeable. Likewise, the feature composition should not be extensible (by adding further, previously omitted features). In dynamic and scripting environments as NX, feature composition is inherently subjected to change. In NX, for example, it would be possible to redefine the product structure and product behavior after restoring the product (`G1`) from its serialized state through reflective operations (such as altering class relations, adding new methods, redefining objects and classes). Evaluating techniques for freezing products at the object and at the class levels are future work (e.g., variants of superimposition based on runtime structures, applying filters to the serialization process).

3.2.5 Implementation

The previous sections presented the NX language-level support for feature-oriented programming and for flexible feature binding strategies in a step-wise manner, introducing the notions of collaboration classes, feature module classes, and extension hierarchy one after another. The full implementation study is given in the Appendix to this paper. We want to stress that this implementation, while not feature complete, is lightweight. The concepts of collaborations and feature modules map to the two metaclasses `Collaboration` and `FeatureModule`. The weaving behavior defined by these metaclasses is implemented by a small code fragment. The code necessary for computing the extension hierarchy fits in 24 SLOC, the code for feature weaving at the class level in 18 SLOC, and its object-level counterpart for weaving at the object level in 20 SLOC. This is completed by further 12 lines for adding some syntactic sugar (e.g., the infrastructure for the managing properties such as `features`). Despite the limitations of SLOC, this weak approximation of code size indicates the low effort required for a basic feature binding framework in NX.

4. RELATED WORK

Compositional approaches [20, 19, 21, 2, 24, 23, 6, 16] to feature-oriented programming (FOP) of software product lines (SPLs) typically support one or several feature binding strategies as defined in Section 2.1. Below, we review the ones which directly influenced our approach. A more complete account on binding support in FOP is given in [19].

Rosenmüller et al. [20, 19] propose code generation from a single asset base integrating both static class-level and dynamic object-level feature bindings in FeatureC++. The framework allows for switching between static class-level and dynamic object-level feature bindings at SPL build time. These assets (class refinements) are organized in a flat folder structure. For static binding, these class structures are merged by superimposition. For dynamic binding, feature classes as part of a GoF Decorator pattern idiom are generated. These feature classes are then organized in decorator chains to implement layered designs, based on method forwarding, using an application-level super-reference list. This entails decomposition issues such as the self-problem. Limitations are due to the host language C++ (e.g., not supporting dynamic class-level composition).

Ostermann [16] puts forth a collaboration-based and layered implementation technique based on prototype delegation (to realize refinement chains) and a variant of virtual classes (to represent collaborations with composition locality ; see Section 2.2). The result allows for dynamic, object-level compositions. Multiple binding schemes are not supported. This delegation layer compares with our dynamic, object-level technique using NX decorator mixins. For example, decorator mixins share the rebinding of the self-reference under delegation.

Smaragdakis and Batory [24, 23] present an implementation technique for collaboration-based designs using mixin layers. Their notions of collaboration-based design and of coarse-grain modularization for step-wise refinements is also a central motivation for FOP in NX. As for the implementation techniques for collaboration-based designs and the notion of mixins, besides C++, Smaragdakis and Batory [24] explore the use of CLOS mixins (i.e., the CLOS variant of multiple class-based inheritance with linearization). Their CLOS implementation study compares with our NX study as NX's OO system is closely modeled after CLOS (e.g., the linearization scheme used). In addition, the CLOS meta-object protocol allows for implementing versatile serializers [18] to be used as outlined in Section 3.2.4 for NX.

In CaesarJ [2] (and the Beta family of languages) the concept of family classes as collections of virtual classes attracted our attention towards the issue of composition locality and influenced the NX implementation of collaboration classes based on constructor generation and object nesting. Also, NX provides comparable means to navigate family classes. The NX helper command `info parent` allows one to access the enclosing object, similar to the pseudo variable `out` in gbeta. Further similarities result from CaesarJ and gbeta composing superclass hierarchies upon binding family classes (and their nested classes) to each other. The nested classes are a variant of abstract subclasses.

In DeltaJ [21] refinements are limited to the class level. A program is generated given a product configuration. We, therefore, classify DeltaJ as a static, class-level approach only. While in our dynamically typed language setting, we stress software compositional issues, Schäfer et al. investigate (static) type safety under feature composition.

In [6], Gunther and Sunkle introduce the Ruby FOP extension rbFeatures. The FOP approach is mainly annotational, that is, feature-specific code is grouped using Ruby blocks (e.g., inside a Proc object) and feature composition is then performed by evaluating an assembled set of such blocks. In our scheme in Fig. 2, this constitutes a composition level distinct from objects and classes which also covers the sub-method level, for example. This meta-programming scheme for script-level composition is implementable in NX. The authors of rbFeatures, however, do not consider object-compositional facilities, in particular Ruby modules. Although missing metaclasses, the techniques introduced in Section 3.2 (with object aggregation) can be approximated using Ruby modules and open class declarations.

5. SUMMARY AND CONCLUSIONS

We presented an approach to dynamic and to static feature bindings, both at the object level and at the class level. The assets of the SPL (the base collaboration and the feature modules) are represented as objects and classes, with collaboration structure being modeled through dynamic object aggregations. The same set of assets is used as the source for dynamic and static feature binding. For the implementation of the approach, we use high-level object-oriented concepts such as multiple class-based inheritance, decorator mixins, metaclasses, object/class aggregation, and object system introspection. The resulting implementation study meets critical requirements, such as providing for a single code base, composition locality, and the avoidance of typical decomposition mismatches in collaboration-based designs. Given appropriate language support as in NX, the approach turns into a lightweight implementation (see the Appendix).

The approach presented is not complete. We have not addressed checking of product line models, evaluating composition constraints (unlike [19]), and handling feature interactions. Also, support for homogeneous and dynamically crosscutting features has not been considered. For the latter, NX provides message-level filters [13]. NX also supports conditional mixins for fine-grained composition control, based on guarding expressions. There are also mixin variants [25] available to enforce strict feature ordering. Besides, while the GPL helps demonstrate similarities and differences to prior work [20, 16], the framework's fit regarding larger-scale SPLs remains to be evaluated.

In a next step, we will extend our feature binding framework beyond structure-preserving binding techniques to support flattening layered collaboration structures (see the merge operator in [15] and traits [4]). This is important to offer optimizations (e.g., minimizing a product's memory footprint) under both the static and the dynamic binding modes, as well as to fully support static feature binding.

6. REFERENCES

[1] S. Apel, D. S. Batory, and M. Rosenmüller. On the Structure of Crosscutting Concerns : Using Aspects or Collaborations ? In *Proc. Workshop Aspect-Oriented Product Line Eng. (AOPLE)*, 2006.

[2] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. An overview of CaesarJ. *T. Aspect-Oriented Software Develop. I*, pages 135–173, 2006.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming — Methods, Tools, and Applications.* Addison-Wesley, 6th edition, 2000.

[4] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits : A Mechanism for Fine-grained Reuse. *ACM Trans. Program. Lang. Syst.*, 28(2) :331–388, 2006.

[5] R. E. Filman, T. Elrad, S. Clarke, and M. Akşit. Aspect-Oriented Programming is Quantification and Obliviousness. In *Aspect-Oriented Software Development*, chapter 2. Addison-Wesley, Oct 2004.

[6] S. Günther and S. Sunkle. rbFeatures : Feature-oriented programming with Ruby. *Sci. Comput. Program.*, 77(3) :152 –173, 2012.

[7] S. O. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *IEEE Computer*, 41(4) :93–95, 2008.

[8] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-oriented Systems. *SIGPLAN Not.*, 21(11) :214–223, June 1986.

[9] R. E. Lopez-Herrejon, D. S. Batory, and W. R. Cook. Evaluating Support for Features in Advanced Modularization Technologies. In *Proc. 19th Europ. Conf. Object-Oriented Programming (ECOOP'05)*, volume 3586 of *LNCS*, pages 169–194. Springer, 2005.

[10] G. Neumann. Dynamic Software Evolution in the Next Scripting Language. http ://next-scripting.org/docs/2.0b3/doc/nx/nx-code-evolution/, last accessed on July 9th, 2012, July 2012.

[11] G. Neumann and S. Sobernig. An Overview of the Next Scripting Toolkit. In *Proc. 18th Annu. Tcl/Tk Conf.* Tcl Association, 2011.

[12] G. Neumann and U. Zdun. Enhancing Object-Based System Composition through Per-Object Mixins. In *Proc. Asia-Pacific Software Eng. Conf. (APSEC'99)*, pages 522–530. IEEE Computer Society, 1999.

[13] G. Neumann and U. Zdun. Filters as a Language Support for Design Patterns in Object-Oriented Scripting Languages. In *Proc. 5th Conf. Object-Oriented Technologies and Syst. (COOTS'99).* USENIX, 1999.

[14] G. Neumann and U. Zdun. Towards the Usage of Dynamic Object Aggregation as a Foundation for Composition. In *Proc. Symp. Applied Computing (SAC'00)*, pages 818–821. ACM, 2000.

[15] H. Ossher and W. Harrison. Combination of Inheritance Hierarchies. In *Conf. Proc. Object-oriented Programming Syst., Languages and Applicat. (OOPSLA'92)*, pages 25–40. ACM, 1992.

[16] K. Ostermann. Dynamically Composable Collaborations with Delegation Layers. In *Proc. 16th Europ. Conf. Object-Oriented Programming (ECOOP'02)*, pages 89–110. Springer, 2002.

[17] S. Rank. *A Reflective Architecture to Support Dynamic Software Evolution.* PhD thesis, University of Durham, UK, 2002.

[18] D. Riehle, W. Siberski, D. Bäumer, D. Megert, and H. Züllighoven. Serializer. In *Pattern Languages of Program Design 3*, pages 293–312. Addison-Wesley, 1998.

[19] M. Rosenmüller, N. Siegmund, S. Apel, and G. Saake. Flexible feature binding in software product lines. *Autom. Softw. Eng.*, 18 :163–197, 2011.

[20] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Code generation to support static and dynamic composition of software product lines. In *Proc. 7th Int. Conf. Generative Programming and Component Eng. (GPCE'08)*, pages 3–12. ACM, 2008.

[21] I. Schaefer, L. Bettini, and F. Damiani. Delta-oriented Programming of Software Product Lines. In *Proc. 10th Int. Conf. Aspect-oriented Software Develop.*, pages 43–56. ACM, 2011.

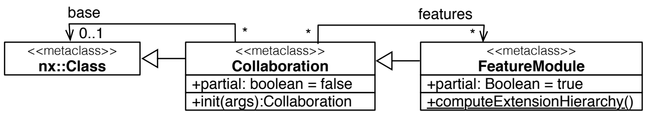
[22] S. Schulze, S. Apel, and C. Kästner. Code Clones in Feature-Oriented Software Product Lines. In *Proc. 9th Int. Conf. Generative Programming and Component Eng. (GPCE'10)*, pages 103–112. ACM, 2010.

[23] Y. Smaragdakis and D. Batory. Mixin Layers : An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM T. Softw. Eng. Meth.*, 11(2) :215–255, 2002.

[24] Y. Smaragdakis and D. S. Batory. Implementing layered designs with mixin layers. In *Proc. 12th Europ. Conf. Object-Oriented Programming (ECOOP'98)*, pages 550–570. Springer, 1998.

[25] U. Zdun, M. Strembeck, and G. Neumann. Object-based and class-based composition of transitive mixins. *Inform. Software Techn.*, 49(8) :871–891, 2007.

Appendix. The NX Implementation Study



```
################################################################################
# Feature Framework Classes
################################################################################
#
# The collaboration metaclass takes a base class and a set of features modules
# to build a new class in its constructor.
#
nx::Class create Collaboration -superclass ::nx::Class {
  :property {base:class [self]}
  :property {features:0..n,type=::FeatureModule ""}
  :property {partial:switch false}

  :public method createAccessors {-context -collaborationClassNames} {
    # Create accessors for the collaboration parts
    foreach name $collaborationClassNames {
      $context public method "new [string tolower $name]" args \
          [subst {${context}::$name new {*}\$args}]
    }
  }

  :public method weave {-baseClass -featureModules -context -partial} {
    set d [::FeatureModule computeExtensionHierarchy \
               -baseClass $baseClass \
               -featureModules $featureModules]
    set collaborationClassNames [dict keys [dict get $d class]]
    if {${:base} ne $context} {
      # Let the product inherit from the extension classes and the base class.
      set superclasses [concat [dict get $d extension $baseClass] $baseClass]
      nsf::relation [self] superclass [concat $superclasses [:info superclass]]

      foreach name $collaborationClassNames {
        # Create child classes as collaboration parts.
        nx::Class create ${context}::$name \
            -superclass [concat [dict get $d extension $name] [dict get $d class $name]]
      }
    }
    if {!$partial} {
      :createAccessors \
          -context $context \
          -collaborationClassNames $collaborationClassNames
    }
  }

  :public method init {} {
    :weave -baseClass ${:base} \
        -featureModules ${:features} \
        -context [self] \
        -partial ${:partial}
  }
}

#
# A FeatureModule is a specialized collaboration.
#
nx::Class create FeatureModule -superclass Collaboration {
  :property {partial:switch true}
  :public class method computeExtensionHierarchy {
          -baseClass:class
          -featureModules:object,type=::FeatureModule,0..n
    } {
    dict set d extension $baseClass ""

    # Create an extension structure for the base class.
    foreach childclass [$baseClass info children -type ::nx::Class] {
      set name [$childclass info name]
      dict set d extension $name ""
      dict set d class $name $childclass
    }

    # For each feature module,
    # (1) add the feature class to the extension list of the base class and
    # (2) create/extend the extension list for the collaboration classes.
    foreach featureClass $featureModules {
      if {[nsf::object::exists $featureClass]} {
        dict set d extension $baseClass \
            [concat [dict get $d extension $baseClass] $featureClass]

        foreach featureChildclass [$featureClass info children -type ::nx::Class] {
          set name [$featureChildclass info name]
          if {[dict exists $d class $name]} {
            # known collaboration class
            dict set d extension $name \
                [concat [dict get $d extension $name] $featureChildclass]
          } else {
            # unknown collaboration class
            set class($name) $featureChildclass
            dict set d class $name $featureChildclass
            dict set d extension $name ""
          }
        }
      }
    }
    return $d
  }
}





################################################################################
# Application Code
################################################################################
#
# A helper class providing the otherwise redundantly implemented "print" method.
#
nx::Class create Printable {
    :public method print {} {
        puts "[self] has vars [:info vars]"
    }
}

#
# A Graph is a collaboration composed of Nodes and Edges.
#
Collaboration create Graph -superclass Printable {
  :property name
  :property {edges:0..n,incremental ""}

  nx::Class create [self]::Node -superclass ::Printable
  nx::Class create [self]::Edge -superclass ::Printable {
    :property from
    :property to
  }
}

#
# Define a feature module "weighted", including a new property "weight" for edges.
#
FeatureModule create weighted {
  nx::Class create [self]::Weight -superclass Printable {
    :property {value 0}
  }
  nx::Class create [self]::Edge {
    :property weight:object,type=::weighted::Weight
  }
  :public method weighted {} {return 1}
}

#
# Define a second feature module "colored".
#
FeatureModule create colored {
  nx::Class create [self]::Color -superclass Printable {
    :property {value 0}
  }
  nx::Class create [self]::Edge {
    :property color:object,type=::colored::Color
  }
  :public method colored {} {return 1}
}

################################################################################
# Code for Object-Level Composition
################################################################################
#
# Extend the Graph collaboration to support object-level compositions.
#
Graph eval {
  # Helper method for providing tailored "new"-methods
  :public method addAccessor {name base mixins} {
    set body "\n    set o \[$base new "
    if {$mixins ne ""} {append body "-mixin [list $mixins] "}
    append body "\]"
    append body {
      foreach {att value} $args {$o [string trimleft $att -] $value}
      return $o
    }
    :public method "new $name" args $body
  }

  #
  # Property "features" is implemented as a slot with its own helper methods. The method
  # "weave" uses the computeExtensionHierarchy method to compute the class dependencies.
  #
  :property features:0..n {
    :method weave {obj featureModules:object,0..n,type=::FeatureModule} {
      set baseClass [$obj info class]
      set d [::FeatureModule computeExtensionHierarchy \
                 -baseClass $baseClass \
                 -featureModules $featureModules]
      set collaborationClassNames [dict keys [dict get $d class]]

      # The following assumes that
      # (1) all mixins are provided by the PL composition and that
      # (2) we can freely overwrite "new *" methods.
      $obj mixin [dict get $d extension $baseClass]
      foreach name $collaborationClassNames {
        $obj addAccessor [string tolower $name] [dict get $d class $name] \
            [dict get $d extension $name]
      }
    }

    :public method assign {obj prop arg} {
      next
      :weave $obj [$obj $prop]
    }
    :public method add {obj prop arg} {
      next
      :weave $obj [$obj $prop]
    }
    :public method delete {obj prop arg} {
      next
      foreach m [$obj info lookup methods -path "new *"] {
        $obj delete method $m
      }
      :weave $obj [$obj $prop]
    }
  }
}
```