Dynamically Adaptable Software Product Lines Using Ruby Metaprogramming

Sebastian Günther and Sagar Sunkle
School of Computer Science
University of Magdeburg, Germany
sebastian.guenther@ovgu.de
sagar.sunkle@ovgu.de

ABSTRACT

Software product lines (SPL) is a paradigm to structure software development assets in a common and reusable form. Out of this common asset base – which includes the application's source code, documentation, and configuration – concrete product variants can be created. The variants are differing in terms of the features, which are basically an increment in functionality important for a stakeholder. Feature-oriented programming (FOP) provides the capability to compose those different variants. In earlier work we presented rbFeatures, a FOP implementation in Ruby. With rbFeatures, features become are first-class entities of the language that facilitate runtime changes of the program. This paper presents an extension to rbFeatures that implements product lines and their variants as first-class entities too. The entities allow powerful runtime-adaptation and configuration, like to add new features or constraints to the product line and the instantiation of several variants with different feature configurations. The particular contributions are to show how Ruby's metaprogramming capabilities are used to design first-class entities and an explanation of the usage of our approach with a common case study.

Categories and Subject Descriptors: D.2.2 [**Software**]: Software Engineering - *Design Tools and Techniques*; D.3.3 [**Software**]: Programming Languages - *Language Constructs and Features*

General Terms: Languages

Keywords

Feature-Oriented Programming, Software Product Lines, Metaprogramming, Domain-Specific Languages, Runtime Adaptation

FOSD'10 October 10, 2010, Eindhoven, The Netherlands

1. INTRODUCTION

In software engineering, features provide additional modularization to applications. Features are special characteristics of software which distinguish members of a program family [2]. Program families evolved to today's understanding of software product lines that share a "common, managed set of features" [26]. In order to provide a more structured approach to software design and implementation, features and other valuable production assets are grouped together to form software product lines [3]. Recently, software product lines with dynamic adaptation facilities are gaining a widespread interest [27, 21, 18]. The adaptability need includes the reconfiguration of variants at runtime and the instantiation of new variants. On motivation for such runtime adaptation is the seamless migration of a 24/7 application when the codebase was updated with new feature code [21].

We see features from a conceptual and an implementation viewpoint. Conceptually, *model features* help to structure the production assets in the form of a product line. At the implementation level, a *concrete feature* is the sum of all code inside a program that belongs to a particular model feature. An important consideration in our research is the idea to provide first-class entities. When features become entities of the program itself, they help to bridge the gap between the conceptual and the implementation level.

In earlier work [8], we presented a FOP implementation in Ruby called rbFeatures. With rbFeatures, developers add feature containments around selected blocks of code. The feature containment condition specify under which feature configuration these parts are active. Only if the condition is satisfied, the contained code will be contained in the application. The functionality provided by rbFeatures includes runtime re-configuration of the application by activating or deactivating features and runtime extension of the program and its features.

Since then we extended rbFeatures. The current version extends rbFeatures with a language for product line modeling. The language's intent is to represent the known tree-like feature models that express an application's model features and their relationships [3], as well as their constraints like mandatory and optional features. By combining rbFeatures with this language, we implemented abstractions for features, product lines, and variants as first-class entities. This particular solution provides rich runtime adaptation and configuration of software product lines, including the provision of multiple variants and variant modification.

This paper provides a complete coverage of feature modeling, feature implementation, and dynamic runtime composi-

tion and modification of a software product line and its variants. The particular contributions are to show how Ruby's metaprogramming capabilities are used to design first-class entities and an explanation of the usage of our approach with a common case study. By describing how we utilized the host language Ruby to build the extension, we wish to show other FOP researchers how to build first-class entities and how this approach supports the goal of features and runtime adaptation.

In Section 2, further background about regarding feature-oriented programming, software product lines, and dynamically adaptable SPL's is explained. Section 3 elaborates the basics of the extended version of rbFeatures, and in Section 4 we explain how first-class variants are implemented as objects and how they can be used for runtime adaptation and customization. Section 5 gives related work and Section 6 summarizes this paper. We apply following formatting: *keywords*, FEATURES, and `source code`.

2. BACKGROUND

2.1 Feature-Oriented Programming

Features can be seen from two perspectives. The fist perspective regards features as all parts of a software that reflect the concerns of a stakeholder [3]. Features are "common aspects [...] as well as differences between related systems" [10]. Features are important in domain engineering to scope the software and they also provide the stakeholder-important requirement of an application. These features are called *model features*.

The second perspective expresses features at the code base – we call them *concrete features*. In this perspective, FOP is a paradigm that was introduced as a "new conceptual model for object and object composition" [20]. It allows grouping and composing sets of classes to obtain different variants of a program. How to implement features or compose variants out of features is an open research field. Since its inception, many FOP implementations have been proposed: Mixin-layers [23], AHEAD-refinements [2], and aspectual feature modules [1] to name a few. In general, the approaches can be divided into two different forms [12]. In the *compositional approach*, features are added as refinements to a base program. The explicit representation of all code belonging to a feature is an expression of the separation of concerns principle [4]. In the *annotative approach*, features are represented as annotations inside the source code. The representation can be implicit on top of the source code like in CIDE [11], which uses a representation on top of the programs abstract-syntax tree, or it can be explicit by using language constructs as in our rbFeatures approach [8].

While both perspectives are certainly providing benefits, the still existing gap between the two representations is to be questioned. We argue that a complete representation of model and concrete features provides the benefits of both worlds: A clean structuring of stakeholder concerns and tooling to prioritize development decisions, as well as the technical capability to build and deliver custom variants with respect to the available configurations as expressed with the software product line model. Section 3 explains how to implement this vision.

2.2 Software Product Lines

When developing software, one fundamental decision is to design either one-off systems or a program family. One-off systems are scoped, configured, and executed for one exact purpose. After its development, the system goes into a stable usage and maintaining period, and eventually is replaced by an successor. On the contrary, program families [4] are applications that are used in different configurations for similar, but not the same purposes. Members of a program family have several *commonalities* with their members while the *variable* part exhibits the different configurations.

Software product lines is the modern name for software families with a special focus on the providing automatic means to derive individual variants from a common codebase. The need for product lines stems from today's strong individualization requirements that drive customization and software flexibility to its height. As [26] explains, "managers must invest strategically in software assets to gain competitive advantage in the battlefield or the marketplace". Following this need, SPL identify, structure, and provide a set of production assets that are systematically reused [3]. The connection between software product lines and features is a compositional one: "product line is a group of products sharing a common, managed set of features" [26]. A feature-diagram can be used to represent the relationships and constraints between the features in a tree-like structure. A particular configuration of features is a valid variant if all the constraints specified in the feature model are satisfied.

Feature-oriented programming is one option to provide the product-line feature structure for the assets, especially for assets related to implementation. Dependent on the particular FOP implementation, this allows different representations of the product line, constraints, features, variants and composition approaches.

2.3 Dynamically Adaptable Software Product Lines

Software product lines with dynamic adaptation facilities are gaining a widespread interest in recent publications [27, 21, 18]. The primary motivation for having runtime adaptation is to provide different variants that support specific application needs. In one case study, complex Enterprise Resource Planning Systems are configured on-site in customer sales acquisition [27]. In the sales dialog, customers express their requirements. The presenter customizes the application accordingly, and the customers can test the application and refine their requirements until they are satisfied. Another use case of dynamic adaptation is to support 24/7 applications [21]. In order to continually evolve the application without providing any downtimes, one approach is to enable the live-update of the running application. Once a new feature has been implemented, the running application is carefully migrated to the new version. In this process, the product line model helps to maintain the structural relationships between the assets and can be used for testing prior to deployment.

3. rbFeatures

rbFeatures [8, 7] enables Feature-Oriented Programming with the Ruby programming language. Features become first-class entities of a program. They are constants that can be used in any expression and are thus open to runtime

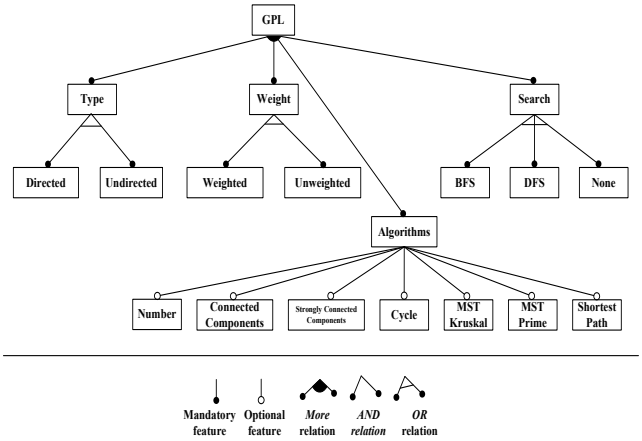

Figure 1: Feature model of the GPL.

modification too.

This section details the basic implementation and application of rbFeatures. Thereby, we use the Graph Product Line (GPL) as the ongoing example. The GPL is a product line that provides different variants for graphs and graph algorithms [15]. We see the tree-like structure of the GPL in ►Figure 1. As can be seen, the product line differentiates the type and weight of a graph, provides search algorithms, and implements numerous other algorithms like determining whether the graph is a connected graph.

3.1 Central Entities

The initial version of rbFeatures consisted of two central entities. The `Feature` module contains all methods that form the public API of features and internal functionality. This module is mixed into normal classes that represent an application's features. The second central entity is the `FeatureResolver`. It defines the required background functionality so that the code associated with a feature or a combination of features is active with regard to the feature configuration.

The extension of a domain-specific language [16] for product line modeling that we developed in an earlier paper [5] adds additional entities. A `FeatureModel` is a configuration unit that represents model features. It contains a name, a list of subfeatures, the position it has in the feature tree, and constraints. The `ProductLine` entity is defined by adding all configured model features. Finally, the `ProductVariant` entities represent a concrete feature-configuration and can be instantiated at runtime.

The relationships between all entities is shown in ►Figure 2. Using the extended version of rbFeatures encompasses the following steps:

1. *Product Line Modeling* – Define the model features, their relationships, and their constraints. Add all model feature to a `ProductLine` object.
2. *Application Implementation* – Implement or feature-refactor an application.
   - Create `Feature` objects that represent the identified the model feature.
   - Form feature containments by enclosing all code parts in a block and provide a containment condition expressing under which feature configuration the containment is executed.
3. *Program Initialization* – Initially execute the program to obtain the first-class representations for the product line, model features, and the concrete features.
4. *Variant Creation and Instantiation* – Create *ProductVariant* objects by configuring a set of activated features that are valid to the feature model, and then instantiate the variant in a defined scope.

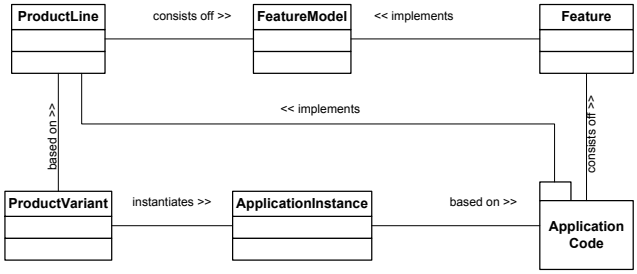

Figure 2: Entity structure of rbFeatures.

The next subsections explain each step in detail.

3.2 Product Line Modeling

The first step is to provide the model features and the productline. Each model feature needs to be declared with the following properties: `name`, position in the tree (`root`, `node`, `leaf`), the `subfeatures`, and a set of constraints (using the keywords `is`, `one`, `more`, `all`, `any`). For example, the declaration of the root node `GPL` takes the following form in ►Figure 3.

This declaration is easy to read for itself. Beginning in Line 2, the name of the feature is defined, its relative position in the feature tree declared, its subfeatures listed, and finally a constraint defined. The constraint expresses that the selection of the GPL feature requires selecting one ore more of it's subfeatures (`Type`, `Weight`, `Search`, and `Algorithms`).

After all features are declared in this way, we can declare the product line object. It uses the syntax shown in ►Figure 4. After a short description in Line 2, the next

```
1 gpl_feature = FeatureModel.configure do
2   name :GPL
3   root
4   subfeatures :Type, :Weight, :Search, :Algorithms
5   requires :GPL => "more :Type, :Weight, :Search,
        :Algorithms"
6 end
```

Figure 3: Configuring the root feature of the GPL.

```
1 GPL = ProductLine.configure do
2   description "The complete GPL"
3   add_feature gpl_feature
4   add_feature type_feature
5   add_feature weight_feature
6   #...
7 end
```

Figure 4: Creating a product line by adding all features.

```
1  class Weighted
2    is Feature
3  end
4
5  class Unweighted
6    is Feature
7  end
8
9  class Undirected
10   is Feature
11 end
```

Figure 5: Implementing basic features.

lines just add the model features to this product line. Once the product line declaration is complete, the `valid?` method checks whether all named subfeatures and features contained in the conditions are included – this helps to detect incorrect product line objects.

After these first steps we have a complete set of model features and the product line available. The next step is to provide the implementation of the product line, and later combine the model and the implementation for creating variants.

3.3 Application Implementation

The next step is to declare the concrete features. Either a complete application including features is developed from scratch, or an existing application is feature-refactored. In both cases, the task is to first define the features entities and then to form feature containments.

As we explained before, concrete features are normal classes. If they do not contain functionality on their own, their declaration is as simple as shown in ►Figure 5. If they contain functionality on their own, then the class declaration body includes additional fields and methods as it is commonly defined in Ruby.

Once the features are defined, we use them to form feature containments. Containments consist of two parts. The first part is called *containment condition*. It is an expression that specifies which activation status one or several features require. Conditions like "If feature A, B and C, but not D are activated", translate to the natural syntax `(A + B + C - D)`. The second part of the feature containment is the *containment body*. It contains code that belongs to the particular feature or the intersection of features specified in the condition. rbFeatures supports three granularity levels of feature code: (i) complete class or module declarations, (ii) method declarations, and (iii) individual lines or even individual characters in lines. The granularity of the containment condition and the containment body enacts high flexibility of declaring code for a single feature or interacting features.

The GPL requires to put both lines containments and complete method bodies in feature containments. As shown in ►Figure 6, the `Weight` feature interacts with `Edge` by defining the local `weight` variable and an accessor for it (Line 2 and 4). The features `Directed` and `Undirected` interact with `Graph` by defining the `directed?` method with a custom body (Line 11–16 and 17–21).

Once the application is completed with all features and feature containments are created, we can initialize the program.

```
1  class Edge
2    Weighted.code { attr_accessor :weight }
3    def initialize(params)
4      Weighted.code { @weight = params.delete :weight }
5      params.delete :weight if params.include? :weight
6      #...
7    end
8  end
9
10 class Graph
11   def initialize(gtype)
12   Directed.code do
13     def directed
14       return gtype == 1
15     end
16   end
17   Undirected.code do
18     def directed
19       return false
20     end
21   end
22   #...
23 end
```

Figure 6: Implementing the GPL application with feature containments.

3.4 Program Initialization

For providing runtime changes to the application, rbFeatures requires a complete representation of the application. Because of the mechanisms used to instantiate a variant in a specific namespace, we use string representation. A `ProcductLine` object stores this string representation and can invoke it several times to define new variants.

Once the code's representation has been provided, the next step is to initially execute it and to define the applications modules, classes, objects, and the features. In this process, we use several hooks and metaprogramming mechanisms that help to change specific parts of Ruby's normal behavior. The most important ones are listed here:

- *Initial execution of feature containments* – Initially all features are deactivated, so that the normal initialization of the program would not execute feature containments. However, whole modules and classes would not be available, and several methods not defined. This could lead to an in-executable program. Because of this, all containment bodies are executed nevertheless, and thus the program is initialized with all entities and methods defined. However, the methods are actually not executable: The method-added hook modifies them.
- *Method-added hook* – Ruby provides several hooks that are called at specific runtime conditions (a complete list is explained in [25]). rbFeatures uses a hook that is triggered whenever a new method is added to an object. The hook checks if there was a feature violation stemming from the last containment condition. If yes, it replaces the method's body with a custom error message specifying the conflicting feature (for example "FeatureNotActivatedError: Feature DFS is not activated").
- *Instantiation prohibition* – Deactivated features are not allowed to create instances. In the initialization phase, we overwrite the `initialize` method to throw an error too, using the explained method-added hook.

3.5 Feature Activation and Deactivation

After initialization, the program consisting of all classes, modules, and methods exist. Yet the provided functionality is limited because all features are deactivated by default. Activating or deactivating features changes the program[1]. Each time a feature changes its activation status, the `FeatureResolver` is triggered to re-evaluate[2] the string representation of the application code. This means to execute all module, class, and method declarations again. Eventually, the containment conditions are now valid, and methods previously not available can now be executed normally.

This modification uses two important metaprogramming capabilities of Ruby: *open classes* and *code evaluation*.

- *Open classes* – Ruby allows modifying all existing entities, even the built-in ones. For example if a method declaration is executed in the same scope and with the same name as an existing method, then the old method is overwritten.
- *Code evaluation* – At runtime, code in the form of `String` or `Proc` objects can be evaluated. Strings are an external format and are slower to evaluate, but using Ruby's string processing capabilities they can also be changed arbitrarily. Procs instead are transformed to an internal representation. Like strings, they can contain any expressions, but are not modifiable after their creation[3] The difference is that procs are similar to closures stemming from functional-oriented programming. Their declaration encloses the state of surrounding variables even if the original context is no longer available.

Both concepts explain the dynamic adaptation capabilities of rbFeatures. At first, the whole application is stored inside a string object, giving full manipulation capabilities of the code with built-in string processing capabilities. Second, containment bodies are actually proc objects, defined with the **do** ... `end` notation shown in previous examples. And third, every time a feature changes its activation status, the complete application is re-evaluated again. Parts of the application that were not available before may get active, and this changes the internal program representation.

3.6 Variant Creation and Instantiation

We consider the case to create a `ProductVariant` object that provides the features DIRECTED, WEIGHTED, DFS, and STRONGLY CONNECTED. The expression to create this object are shown in ►Figure 7. The variant receives a name, a parameter pointing to the `ProductLine` model, and in its body various features are activated. When the variant is created, it is automatically added to the product line and can be retrieved from there.

[1]In the current version of rbFeatures, manual configuration using the first-class feature objects is still possible. But this should be used with care because no product line model is available and therefore the feature configuration is not checked. This could lead to buggy programs. Therefore, the use of `ProductVariant` objects as explained in the next subsection 3.6 is recommend.

[2]The method `eval` is used to execute `String` or `Proc` representations of Code, hence we speak of evaluation.

[3]At least with the standard library. In [7] we showed how an external library can be used to obtain a string representation of a a proc, to modify it, and to writ it back to the proc.

```
1 ProductVariant.configure
2   :name => "SimpleVariant",
3   :pl => GPL do
4     activate_features :Directed,
5                       :Weighted,
6                       :DFS,
7                       :Strongly_Connected
8   end
9 end
```

Figure 7: Configuration of a product variant.

Once the variant is available, a call to it's `instantiate!` method actually creates an instance of this variant. This triggers the following steps:

1. Check whether the configured features are valid to the product line model by checking that all specified constraints are satisfied.
2. Compose a string template consisting of a module and the product line code. The module uses the configured name of the variant and serves as a namespace.
3. Add the string template to the core entity `FeatureResolver` (see Section 3.1).
4. Initialize the application by evaluating the template once.

From this moment on, the code contained in the variant is available at its separate scope, and all feature activation changes are governed by the `FeatureResolver` and the `ProductLine` model. Whenever a feature's activation status is changed and possibly a variant's instance modified, the new configuration is checked with the product line model. Only valid feature configurations are allowed. When variants are used, rbFeatures synchronizes the various methods that change a feature's activation status with each other. For example, changing the feature status in the variant or adding a new feature is immediately reflected in the instance. Also, changes of features directly in the variant instance also synchronize with the variant object.

Now we will see the application of these changes in an example.

4. RUNTIME ADAPTATION EXAMPLE

In this section we give an example on how to use the facilities of rbFeatures for providing multiple variants and runtime adaptation.

We assume the GPL example is completely implemented in terms of the product line model, the application code, and concrete features. Then two variants are created using the `ProductVariant` entity in ►Figure 8, Line 1–15. The `ShortestPathVariant` includes the features WEIGHTED, DIRECTED, and SHORTEST PATH, and the *DFSVariant* Includes the features WEIGHTED, DIRECTED, and DFS.

The next step is to instantiate the variants and to create graphs inside them. To create the instance and a graph object for the `ShortestPathVariant`, the expressions in ►Figure 8 (Line 17–29) are used. First, the variant needs to be instantiated with the `instantiate!` method. Second, we select the variant by using the `variant` method of the product line object. Third, a code block is executed in Line 10–28 to create the graph. The `DFSVariant` is created similarly in ►Figure 8 (Line 33–42), but it receives a simpler graph. The resulting

```
ProductVariant.configure
  :name => "ShortestPathVariant",
  :pl => GPL do
    activate_features :Weighted,
                      :Directed,
                      :ShortestPath
  end
end

ProductVariant.configure
  :name => "DFSVariant",
  :pl => GPL do
    activate_features :Weighted,
                      :Directed,
                      :DFS
end

GPL.variant("ShortestPathVariant").instantiate!

ShortestPathVariant.class_eval do
  graph = Graph.new
  1.upto(6) { |n| graph + node(n) }
  graph + edge(1 => 2, :weight => 1)
  graph + edge(1 => 3, :weight => 2)
  graph + edge(2 => 5, :weight => 4)
  graph + edge(2 => 4, :weight => 2)
  graph + edge(4 => 6, :weight => 12)
  graph + edge(3 => 6, :weight => 22)
  SPgraph = graph
end

GPL.variant("DFSVariant").instantiate!

DFSVariant.class_eval do
  graph=Graph.new
  1.upto(6) { |n| graph + node(n) }
  graph + edge(1 => 2, :weight => 1)
  graph + edge(2 => 3, :weight => 2)
  graph + edge(2 => 4, :weight => 3)
  graph + edge(3 => 5, :weight => 4)
  graph + edge(4 => 5, :weight => 4)
  DFSgraph = graph
end
```

Figure 8: Creating two distinct variant objects and instantiating them.

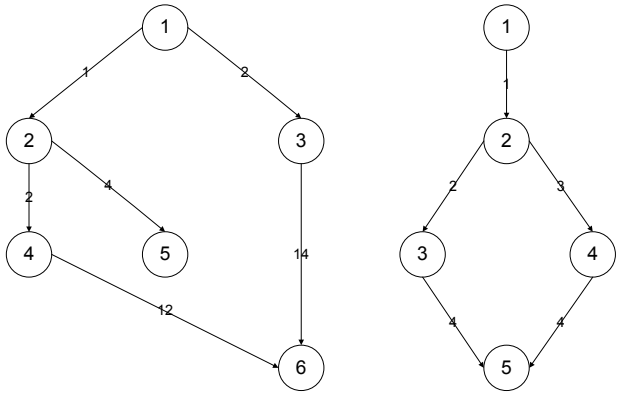

Figure 9: Resulting graph structures.

```
ShortestPathVariant.eval { SPgraph.length } # => 6
DFSVariant.eval { DFSgraph.length } # => 5

ShortestPathVariant.class_eval do
  shortest_path(SPgraph, 6) # => [1,2,4,6]
end

DFSVariant.class_eval do
  shortest_path(DFSgraph, 5) # =>
       FeatureNotActivated Error: Feature
       ShortestPath is not activated
end

GPL.variant("DFSVariant").activate_features
    :ShortestPath

DFSVariant.class_eval do
  shortest_path(DFSgraph, 5) # => [1,2,3,5]
end
```

Figure 10: Runtime adaptation of `ProductVariant` objects.

graphs are shown ►Figure 9 – `ShotestPathVariants` to the left, and `DFSVariant` to the right.

Now that both variants are created and instantiated, we can use and modify them at will. In ►Figure 10, following modifications are applied.

- Test whether each variant has an independent graph object (Line 1 and 2).
- In the `ShortestPathVariant`, calculate the shortest path between the node 1 and 6 and show the result (Line 5).
- Also, try to calculate the shortest path between node 1 and 5 in the `DFSVariant` (Line 9). But this raises an error, demanding that the SHORTESTPATH feature needs to be activated in this variant first.
- Activate the SHORTESTPATH feature (Line 12).
- Now the shortest path can be calculated and the result shown (Line 15).

There is no limit to the number of product lines, variants, and variant instances that can be created (except physical borders like available memory). Also, the first-class objects can be changed at will, including runtime updates of the product line model, which are reflected back down to the instances. Here are some more examples how to use the flexibility given by the first-class entities:

- Define new model features and add them to a product line.
- Inside a variant, add a new block of code that contains a new concrete feature.
- Offer different variants to the user and record which feature combinations are heavily used. Use this information to prioritize development of additional features.
- Analyze how a particular variant is used, change its implementation at runtime by activating a new feature and changing the instance.

5. RELATED WORK

In the research field of dynamic runtime adaptation, several examples and approaches have been presented. The following subsections explain general frameworks, implementation mechanism, and applications.

General Frameworks

One general framework for runtime adaptation expresses three important concerns: (*1*) explicit architectural model, (*2*) provision of structural and behavior constraints, and (*3*) the availability of software connectors for runtime changes

[18]. In rbFeatures, concerns 2 and 3 are expressed with the help of the software product line constraints and the explained concepts of open classes and runtime code evaluation. Furthermore, concern 1 is of no importance in rbFeatures because the feature containments are applicable to any source code independent of its place in the software architecture.

Another work suggests an application-independent and generic meta-model for runtime adaptation [17]. The model considers systems operations, services, and ports of an application as possible entry points for changes, and explains that a concrete binding and implementation for each one can support the adaptation. This generic model can explain how several concrete adaptation approaches work. Recently, rbFeatures was applied to web applications where different feature configuration determine the available functions and web-pages [6]. Altering the behavior of HTTP request handlers and the offered ports is similar to providing a service, and thus rbFeatures can be seen as lightweight instance of this approach too.

Finally one model actively suggests to use features as the dominant entities that drive runtime adaptation [14]. The paper introduces a feature model, binding units, and a feature binding graph which is used to backup the stepwise change of the current features configuration. This is supported by rbFeatures too, albeit the changes in terms of checking the feature configuration with the product line model are comparatively simple.

Implementation Mechanisms

Looking at concrete implementation mechanisms, [22] explains an approach how to enable features for static and dynamic binding. Static binding is a a-priori deployment decision that confines some parts of the application to one specific configuration. Dynamic binding flexibilizes the development of features on the one hand, but it introduces additional overhead like memory consumption or runtime performance degradation. The paper shows how to use the delegator pattern and refinements using binding units can combine both approaches. This is an interesting idea to provide other mechanisms in Ruby as well. Although there is not compile time in Ruby, C extensions to the interpreter could be written that allow the pre-configuration of an application.

Another approach uses Java in order to adapt a software product line and its variants at runtime [21]. Two mechanisms to add new code based on a changed feature model. At first, existing classes are replaced with a similar, slightly evolved class but with another name through the classloader. Second, Java HotSwap is applied to change all method calls that have a callee which is the modified class. How this specific technique can be used for Feature-Oriented Programming in Java is shown in [24]. Ruby supports such runtime modification and metaprogramming mechanisms out of the box, so rbFeatures just needs to use the existing mechanisms.

Applications

Finally considering concrete applications, we see that [27] explains a plug-in based adaptation mechanism on top of the .NET platform. In this paper, some scenarios where runtime adaptation is required or beneficial are explained. One example is a live sales presentation of an enterprise resource planning system. At the presentation, the system is dynamically configured according to obtain the best feature combination which satisfied most or all requirements of the customer. Another example explains a product line which is customized in accordance with the physical and execution environment [19]. The paper presents a case study in which an application displays information about movies. The application features a cache which is activated once the bandwidth of an internet-connection reaches a certain threshold. From thereon, the application serves the data out of the cache instead of live from the server.

6. SUMMARY

This paper explained how feature-oriented programming and runtime adaptation of variants can be achieved by using first-class entities. The approach uses Ruby as the implementation language. By using existing objects (classes and modules), metaprogramming capabilities (open classes, runtime code evaluation, hooks) and functional programming (support of closures as anonymous code blocks), powerful first-class representations of product lines, features, and variants can be created. This approach is generalizable to bring runtime adaptation support for product line variants to other applications as well. Provided the chosen host languages supports similar mechanisms, first-class entities can be created in other languages as well.

A future research direction is to climb the ladder of available abstractions even higher. Once features, product lines, and variants become first-class entities of a host language, they build a reflexive layer about applications. The applications are becoming an abstraction which enables fine-grained modifications and runtime adaptation. By extending the reflexive layer with additional concerns and paradigms, such as aspect-oriented programming [13] or context-oriented programming [9], whole systems comprising several applications can be expressed with powerful meta-expressions. This idea can be easily visualized as building a product line that consists of other product lines. A system as a whole is adaptable by modifying its components which are variants of individual product lines. This architecture can be used to react to complex environmental changes.

Acknowledgments

We thank the anonymous reviewers for helpful comments on an earlier draft of this paper.

7. REFERENCES

[1] S. Apel. *The Role of Features and Aspects in Software Development*. Dissertation, Otto-von-Guericke-Universität Magdeburg, 2007.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197. IEEE Computer Society, 2003.

[3] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, San Franciso et al., 2000.

[4] E. W. Dijkstra. *Notes on Structured Programming*. Academic Press Ltd., London, 1972.

[5] S. Günther. Engineering Domain-Specific Languages with Ruby. In H.-K. Arndt and H. Krcmar, editors, *3. Workshop des Centers for Very Large Business*

Applications (CVLBA), pages 11–21, Aachen, 2009. Shaker.

[6] S. Günther. Multi-DSL Applications with Ruby. *IEEE Software*, 27:25–30, 2010.

[7] S. Günther and S. Sunkle. Enabling Feature-Oriented Programming in Ruby. Technical report (Internet) FIN-016-2009, Otto-von-Guericke-Universität Magdeburg, 2009.

[8] S. Günther and S. Sunkle. Feature-Oriented Programming with Ruby. In *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*, pages 11–18, New York, 2009. ACM.

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

[10] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[11] C. Kästner and S. Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, Sept. 2009.

[12] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320, New York, 2008. ACM.

[13] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Berlin, Heidelberg, New York, 1997.

[14] J. Lee and D. Muthig. Feature-Oriented Analysis and Specification of Dynamic Product Reconfiguration. In *Proceedings of the 10th Internationale Conference on Software Reuse (ICSR)*, pages 154–165, Berlin, Heidelberg, 2008. Springer Verlag.

[15] R. E. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Productline Methodologies. In *Proceedings of the Third International Conference on Generative and Component-Based Software Engineering (GPCE)*, volume 2186 of *Lecture Notes in Computer Science*, pages 10–24, Berlin, Heidelberg, Germany, 2001. Springer Verlag.

[16] M. Mernik, J. Heering, and A. M. Sloane. When and How to Develop Domain-Specific Languages. *ACM Computing Survey*, 37(4):316–344, 2005.

[17] B. Morin, O. Barais, and J. Jézéquel. K@rt: An Aspect-Oriented and Model-Oriented Framework for Dynamic Software Product Lines. In *Proceedings of the 3rd International Workshop on Models@run.time*, pages 127–136, 2008.

[18] P. Oreizy, N. Medvidovic, and R. N. Taylor. Runtime Software Adaptation: Framework, Approaches, and Styles. In *Companion of the 30th International Conference on Software Engineering (ICSE2010)*, pages 899–910. ACM, 2008.

[19] C. A. Parra, X. Blanc, and L. Duchien. Context Awareness for Dynamic Service-Oriented Product Lines. In D. Muthig and J. D. McGregor, editors, *Proceedings of the 13th International Conference on Software Product Lines (SPLC)*, volume 446 of *ACM International Conference Proceeding Series*, pages 131–140. ACM, 2009.

[20] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Berlin, Heidelberg, Germany, 1997. Springer Verlag.

[21] M. Pukall, N. Siegmund, and W. Cazzola. Feature-Oriented Runtime Adaptation. In *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution @ runtime*, pages 33–36, New York, NY, USA, 2009. ACM.

[22] M. Rosenmüller, N. Siegmund, G. Saake, and S. Apel. Combining Static and Dynamic Feature Binding in Software Product Lines. Technical Report FIN-013-2009, Otto-von-Guericke-Universität Magdeburg, 2009.

[23] 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*, 11:215–255, 2002.

[24] S. Sunkle and M. Pukall. Using Reified Contextual Information for Safe Run-time Adaptation of Software Product Lines. 2010.

[25] D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby 1.9 - The Pragmatic Programmers' Guide*. The Pragmatic Bookshelf, Raleigh, USA, 2009.

[26] J. Withey. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI96-TR-010, Software Engineering Institute, Carnegie Mellon University, 1996.

[27] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer. Supporting Runtime System Adaptation through Product Line Engineering and Plug-in Techniques. In *Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS)*, pages 21–30. IEEE Computer Society, 2008.