3 Supporting Heterogeneous Configuration Knowledge of Software Product Lines with Domain Knowledge Modeling Languages

Using object-oriented frameworks as bases for building software product lines might bring several benefits. The main advantage of frameworks is their built-in functionalities. Instead of understanding the implementation of the framework, developers should only keep focus on the ways of configuring and customizing concepts. The framework programming interface is the responsible for providing the set of supported concepts and constraining the means of instantiating them in the source code. However, understanding, verifying, and evolving, framework-based source code are challenge (Antkiewicz and Czarnecki 2006). As a consequence, this might increase complexity and cost of adopting framework-based product lines following an extractive approach (see Chapter 2).

In software engineering, modeling is a well recognized technique for dealing with complexity. A model is a representation of a system that is useful for investigating the properties of entities, phenomena, or process that are part of the system. In some case, one model allows for answering questions about the system or predicting future outcomes. Models come in different flavours. A UML model, a Java program, a XML document, an entity-relationship schema are all examples of models. Every model needs to be expressed using a formal or informal meta-modeling language. A meta-modeling language is a set of models that can be expressed using a collection of concepts and their mutual relations within a certain domain. For example, a XML Schema defines all elements and grammatical rules governing the order of elements that the XML document content must satisfy.

In this Chapter, we present a domain-specific model-supported engineering of software product lines that intends to deal with the complexity and cost of adopting framework-based product lines following an extractive approach. It is based on domain knowledge modeling languages, which formalizes framework's programming interface and concepts. We propose using eCore as abstract syntax definition formalism. We choose to define the structure of framework's programming interface using eCore because it is convenient for expressing well-formedness rules based on multiplicities and parameterized types, which is an important aspect of DKMLs. eCore is a metamodeling notation equivalent to the Essential Meta-Object Facility (eMOF) profile. The Meta-Object Facility is a Object Management Group (OMG) standard for expressing metamodels of modeling languages. It is heavily used in industry today, and most of modeling tools rely on MOF as metamodeling language.

In the following sections we explore the details of engineering software product lines with DKMLs. We define the DMKL concept and its properties in Section 3.2. Then, we discuss how feature visualization occurs with DKMLs through examples in Section 3.3. As DKMLs must capture the knowledge about feature assignment to concept instances, in Section 3.4 we make model well-formedness checking feature sensitive. We define some well-formedness rules that guide the developer on creating models that always conform to the abstract syntax of the DKMLs, that is, models that always meets the framework's programming interface. Once the models are well-formed, we use a constraint satisfaction programming model to ensure that every feature assignment are respecting both feature model and DKMLs semantics.

3.1 Domain-specific model-supported Engineering of Framework-based Software Product Lines

Software product line is *a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way* (Clements and Northrop 2001). Software product line engineering (Clements and Northrop 2001) is a paradigm for constructing, customizing and delivering products from a set of reusable artefacts. We consider in this work feature-oriented software product line engineering (Apel and Kästner 2009, Czarnecki and Eisenecker 2000). The process of engineering product lines comprise four general phases according to (Pohl et al. 2005, Czarnecki and Eisenecker 2000): (i) domain analysis; (ii) domain design; (iii) domain implementation; (iv) product configuration.

The domain analysis is the phase of defining the product line scope in terms of features. A feature (Czarnecki and Eisenecker 2000, Apel and Kästner 2009) is a system property that is relevant to any stakeholder (analysts, architects, managers, developers). Therefore, the feature concept pervades all of other phases. There are several classifications for features (Czarnecki and Eisenecker 2000). More common classification classifies features as: mandatory, optional, alternative, and or-feature. Feature cardinality, groups and attributes represent the variability. The cardinality determines the number of times the feature should occur in one product. The group expresses choices among features. Finally, attributes represent a numeric or textual value of a feature. Features are hierarchically organized into feature diagrams, where each successively deeper level in the tree corresponds to a more fine-grained configuration option. Feature models also provide additional specifications of interaction among features. Interactions are defined as parent-children or cross-tree relationships. Interaction captures constraints that must be held when selecting a set of features during the product configuration phase. Feature are organized in a feature model, which are used to distinguish the software product line commonalities from product-specific features (optional/alternatives)

In the design and implementation phases, the developers typically create a set of reusable artifacts and the configuration knowledge. The configuration knowledge relates features and reusable artifacts. The configuration knowledge might exist in different flavours, but in general it specifies which artefacts implements combination of features. More important, the configuration knowledge also establishes the valid relation among the reusable artefacts implementing features. Based on the feature model and configuration knowledge, products can be derived in most of cases automatically. Due to the configuration knowledge, the developers only needs to specify the set of features desired for the intended product.

To address the heterogeneous configuration knowledge problem we abandon annotation-based and general-purpose model-based techniques and instead use domain knowledge modeling languages to represent the configuration knowledge. We propose a development model that is a extension of traditional code-oriented techniques. In this development model, during the design and implementation phases, developer still identifying features in existing source code, however now they have an option of explicitly identify feature as domain concepts. For that, they instantiate a domain-specific model by creating visual representations of existing concept instances. The model describes the product line from the framework perspective and instantiation constraints. To assign features to concept instances, a developer selects the respective elements in domain knowledge models and expresses this directly in the models.

The domain-specific models are expressed using DKMLs. The metamodel of an DKML is needed for the correct creation of model elements and the assignment of features to concept instances. Unlike in model-driven engineering, domain-specific models are auxiliary development artifacts, which enables easy adoption in existing projects. Therefore, it can be considered as an unified technique to product line development. Domain-specific models and source code coexist and build on each other instead of opposing each other. As a consequence, domain knowledge models can be (semi-)automatically projected from the existing code and as benefits they provide: (i) a visual representation of the software product line from the framework perspective; (ii) easy navigation from model to source code instantiating concepts; and (iii) checking the conformance of configuration knowledge against the framework's programming interface.

In addition to feature model and domain-knowledge models, the implementation model is employed to be a visual representation of source code, in terms of programming languages concepts such as packages, classes, files. This model also represents fine-grained references to fragments inside source code artifacts. Finally, the configuration model organizes the assignments among features and elements from the domain knowledge models and/or implementation model.

3.2
DKML: Definition and Properties

A DKML is a language designed for specifying the configuration knowledge of a single domain. Indeed, heterogeneous programming interfaces are the basis of many modern software development principals: service oriented architecture or component-based development. Heterogenous API integration is role in the construction of new web-based software applications, for example to support interwind data and functional code from different sources (e.g., only communities, on demand video systems, so on). More generally, building any medium or even small size enterprise software application usually involves managing a plethora of programming interfaces. Therefore, in complex software product lines, multiple DKMLs might be necessary to cope with different concerns (e.g., service integration, business process, user interface).

Each DKML consists of an abstract syntax complemented with two dimensional relations: the first dimension corresponds to associations and references between DKML concepts; the second dimension expresses code mappings, that is, they hold references to source code artifacts instantiating concepts. The first dimension – concept's associations and references – captures the framework's programming interface in terms of concept decomposition. The second dimension – code mappings – expresses the instantiation constraints, that is, the steps that developers must fulfil when they are instantiating concepts.

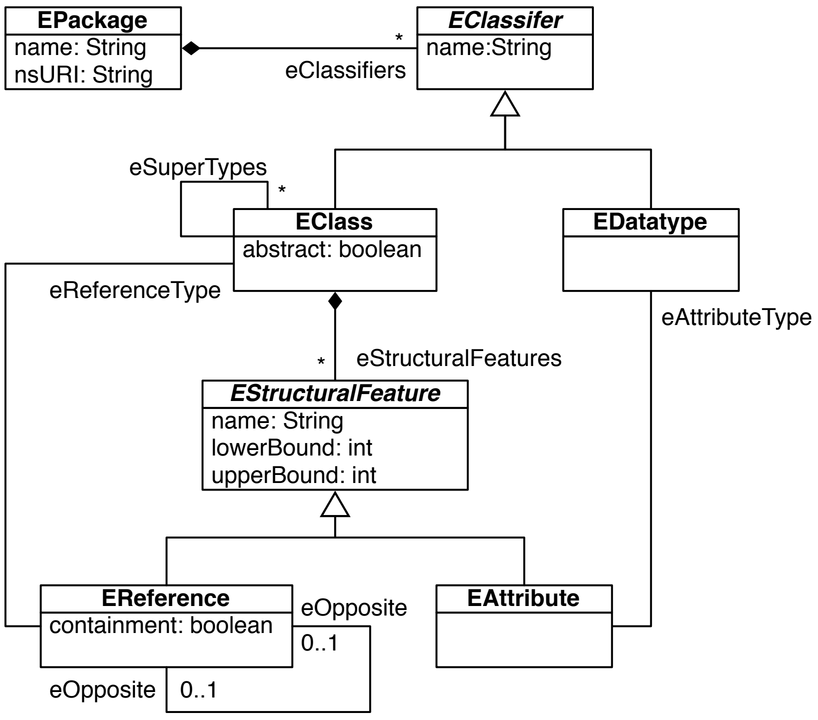


Figure 3.1: Ecore language structure.

The abstract syntax defines a decomposition of framework-provided concepts into associations and references between `class` abstraction, and resemble the Ecore language structure (see Figure 3.1). The Ecore syntax encompasses the following elements: EPackage, EClass; EAttribute; EReference; EDataType. Every Ecore model must contain a root element representing the entire model. The model has children elements that are represented as EPackages, whose children are EClasses. Children of classes are EAttributes or EReferences. Therefore, the DKMLs are rendered hierarchically with a focus on the concept decomposition and towards to showing relations among concepts. An important property of DKMLs is the support for expressing optionality, that is, it provides means to express where a concept instance can be optionality removed from a domain knowledge model. This mechanism guarantees that no product can invalidate the DKML and consequently the framework's programming interface.

In the DKML abstract syntax `class` represents elements with a name `attribute` and zero or more `references`. `References` represent one end of an association between two classes. It can be a simple containment – association – or a parametrized containment – reference – to the `class` to which it points. Each `reference` has a cardinality in the form `<min,max>`, which is an interval specifying how many times a concept instance can or must be created. Note that a name `attribute` is essential for defining concept instances in DKMLs.

Spring Domain Knowledge Modeling Language

```
[DKML] Spring
   <1-*> [reference] context : Context
[class] Context : Selectable
   <1-*> [reference] bean : Bean
[class] Bean : Selectable
   <1-1> [reference] interface : BeanInterface
   <1-*> [reference] implementation: BeanImplementation
[class] BeanInterface
[class] BeanImplementation : Selectable
   <1-*> [reference] constructorInjection : ConstructorInjection
   <1-*> [reference] propertyInjection : PropertyInjection
[class] ConstructorInjection : Selectable
   <1-1> [reference] constructorParameter : ConstructorParameter
   <1-1> [reference<Bean>] bean : Bean
[class] ConstructorParameter
[class] PropertyInjection : Selectable
   <1-1> [reference] propertyMethod : PropertyMethod
   <1-1> [reference<Bean>] bean : Bean
[class] PropertyMethod
```

Figure 3.2: Exemplar Spring Domain Knowledge Modeling Language.

Instead, any other attribute is required. `Attributes` do not take part in the configuration knowledge. The abstract syntax also defines the `selectable`, which serves to give configuration foundation for DKMLs. While the `class` determines the universe of concepts accountable to have those instances present in every model created from a DKML, the `selectable` takes place to define which concepts may have those instances optionally excluded. Therefore, according to DKML semantics, only instances of `selectable` concepts are candidate to be assigned to features in the configuration knowledge.

Figure 3.2 illustrates a DKML for Spring framework. This language expresses the decomposition of Spring-provided concept into `classes` and `references`. The *Context* element is a `class`. It holds zero or more associations to *Bean*, which is also a `class`. The *Bean* in turn holds one and only one reference to *BeanInterface* `class` and one or more references to *BeanImplementation* `class`. The *BeanInterface* means the interface that is implemented by its *Bean*. The *BeanImplementation* represents the code implementing the services of its associated *Bean*. Note that in this case *Beans* might provide more than one choice of implementation for one defined interface, although one and only one will be part of a product. The *BeanImplementation* holds references to `class` elements meaning the different variants of dependency

Spring Domain Knowledge Model

```
DKM Spring
   Context : application-context
      Bean : WeatherService
         BeanInterface : WeatherService
         BeanImplementation : WeatherServiceImpl
            ConstructorInjection : CityDAO
               ConstructorParameter : cityDao
               Reference<Bean : CityDAO>
            ConstructorInjection : WeatherUserServiceDAO
               ConstructorParameter : weatherUserServiceDAO
               Reference<Bean : WeatherUserServiceDAO>
      Bean : CityDAO (...)
      Bean : WeatherUserServiceDAO (...)
```

Figure 3.3: Exemplar Spring Domain Knowledge Model.

injection provided by Spring. The *ConstructorInjection* `class` represents the constructor-based variant (see Section 2.1.1). It contains a reference to one and only one *ConstructArgs* `class` representing to which constructor parameter the concept instance refers. The *PropertyInjection* `class` represents the setter-based variant (see Section 2.1.1). It contains one and only one reference to another `class` called *PropertyMethod*, which represents the methods defined in the code customization used by Spring to dependency injection.

A DKM conforming to this abstract syntax describes Spring-provided concept instance. In Figure 3.3 is defined a concrete *Context* – `applicationContext` – comprising series of *Beans* – `WeatherService`, `CityDAO` and `WeatherUserServiceDAO`. The Bean `WeatherService`, for example, holds a reference to a concrete *BeanInterface* called `WeatherService`. It also encompasses a reference to a concrete *BeanImplementation* called `WeatherServiceImpl`. The *BeanImplementation* in turn defines two concrete *ConstructorInjection* – `CityDAO` and `WeatherUserServiceDAO`. Each *ConstructionInjection* element defines its respective constructor argument in the code customization of the *Bean* `WeatherService`. Finally, each *ConstructorInjection* – `CityDAO` and `WeatherUserServiceDAO` – also maintains a reference for injectable *Bean*, `CityDAO` and `WeatherUserService`, respectively.

In this context, the abstract syntax captures the concepts as `classes` and encodes the framework's programming interface through `references` and the configuration semantic as `selectabel`. For example, the Spring DKML ensures that every *Bean* instance explicitly defines its interface and at least one implementation. Note that the *BeanInterface* `class` cannot be assigned

Spring Domain Knowledge Modeling Language

```
[DKML] Spring
   <1-*> [reference] context : Context
[class] Context : Selectable
   <1-*> [reference] bean : Bean
   <1-1> [reference<File>] context : File
[class] Bean : Selectable
   <1-1> [reference] interface : BeanInterface
   <1-*> [reference] implementation: BeanImplementation
[class] BeanInterface
   <1-1> [reference<Class>] interface : Class
[class] BeanImplementation : Selectable
   <1-*> [reference] constructorInjection : ConstructorInjection
   <1-*> [reference] propertyInjection : PropertyInjection
   <1-1> [reference<Fragment>] configuration : Fragment
   <1-1> [reference<Class>] implementation : Class
[class] ConstructorInjection : Selectable
   <1-1> [reference] constructorParameter : ConstructorParameter
   <1-1> [reference<Bean>] bean : Bean
   <1-1> [reference<Fragment>] constructorArgs : Fragment
[class] ConstructorParameter
   <1-1> [reference<Fragment>] parameter : Fragment
[class] PropertyInjection : Selectable
   <1-1> [reference] propertyMethod : PropertyMethod
   <1-1> [reference<Bean>] bean : Bean
   <1-1> [reference<Fragment>] property : Fragment
[class] PropertyMethod
   <1-1> [reference<Fragment>] setProperty : Fragment
   <1-1> [reference<Fragment>] getProperty : Fragment
```

Figure 3.4: Spring Domain Knowledge Modeling Language with references.

to features. This obeys the Spring instantiation constraint that asks for an interface for every *Bean*. Rather, the *Bean* is defined as `selectable`, meaning that a *Bean* instance can be defined and exclude in any product.

The mapping of the abstract syntax to source code artifacts defines how concepts map to code configuration and customization. As concept instance code might consist of multiple artifacts, mappings of a single Class might correspond code scattered across the product line source code. Class might also represent semantics facts about the framework. In this case, `Class` might not correspond to any code, and consequently not expresses any code mapping. Code mappings are parametrized `references` in the abstract syntax structure holding references to one element from the implementation model. That is, each mapping has a predefined type. It must refer to *Classes*, *Files*, *Folders*, *Packages*, or *Fragments*. That way, DKMLs are not fixed to a particular artifact type and provides support for a new set of artifact types. For example, the implementation model can be modified to visually represents C++ artifacts, such as, header files. Code mappings in the abstract syntax must be defined as `references` having cardinality of one and only one – `<1-1>`, that is, every `class` instance directly corresponds to a source code element.

In Figure 3.4, mappings are attached to `class` as parametrized `references`. For example, the mapping *context* attached to the *Context* `class` specifies that instances of such concept correspond to *Files*. The mapping *class* attached to *BeanImplementation* `class` specifies that its instances correspond to *Classes*. More fine-grained mappings can also be specified via *Fragments*. For example, the mapping *constructorArgs* attached to *ConstructorInjection* `class` specifies that instance such concept correspond to *Fragments*, in this case, the `construction-arg` tag defined in an XML application context files (see Section 2.1.1). Note that semantic mapping that specifies the actual correspondence cannot be defined directly as `classes` and `references`. As we will show in next Chapter, the abstract syntax needs to be enriched with some metadata about code patterns in order to support the actual correspondence of `classes` and `references` to source code elements.

Figure 3.5 presents mapping definitions for the DKML from Figure 3.3. The instance *application-context* of the *Context* `class` directly corresponds to *File* `applicatonContext-userservices.xml`. An instance

Spring Domain Knowledge Model

```
DKM Spring
    Context : application-context
        Reference<File : applicationContext-userservices.xml>
        Bean : WeatherService
            BeanInterface : WeatherService
            BeanImplementation : WeatherServiceImpl
                Reference<Fragment : <bean id="WeatherService(…)>
                Reference<Class : WeatherUserServiceImpl.java>
                ConstructorInjection : CityDAO
                    ConstructorParameter : cityDao
                    Reference<Bean : CityDAO>
                ConstructorInjection : WeatherUserServiceDAO
                    ConstructorParameter : weatherUserServiceDAO
                    Reference<Bean : WeatherUserServiceDAO>
        Bean : CityDAO (...)
        Bean : WeatherUserServiceDAO (...)
```

Figure 3.5: Spring Domain Knowledge Model with references.

of *BeanImplementation* `class` corresponds exactly to one code configuration defined by the concrete *Fragment* `<bean id="WeatherService" class="...WeatherServiceImpl>"` and to one code customization defined by the *File* `WeatherUserServiceImpl.java`. Note that code mappings strictly express the steps that developers must fulfill to properly instantiate the correspondent framework-provided concept. Therefore, as we will discuss in the next sections, they directly define how concepts realize their features, further improving the visualization of features code, in addition to enable automated product derivation, consistency checking and guided development.

3.3 Visualizing Features Code

Program comprehension has become an important concern of software development. Especially in the case of software product lines building via a extractive/reactive fashion, the source code is constantly modified and grow larger, which means that a great deal of effort is spent on performing evolution tasks. Specially, understating software product line source code often requires the analysis of individual feature codes apart from the base code, as well as their interactions. In Chapter 2, we criticized code-oriented techniques for their suboptimal support for visualizing feature code in framework-based software product lines, missing direct traceability based o framework-provided concepts and tendency to obfuscate and scatter feature code along the entire base code.

In previous section we integrated DKMLs into a traditional product line engineering (Apel and Kästner 2009, Czarnecki and Eisenecker 2000) to encapsulate and document the configuration knowledge using domain-specific concepts and their instantiation constraints. In this section, through the previously presented illustrative examples, we show how DKMLs can enhance the visualization of feature code. We distinguish two means of visualizing feature code:

– *Visualizing on domain knowledge models* abstracts the source code and support developers to comprehend the configuration knowledge based on framework-provided concepts and their associations.

– *Visualizing on source code* is the basic way to understand assignments of features to source code, since its elements are shown annotated with the according feature. Visualizing on source code mimics the annotation-based techniques (see Section 2.2.1).

Visualizing on domain knowledge models Observe in the examples that one of the key facilities of DKMLs is that developers can find what concept instances belong to a feature without being distracted by other concerns. Domain knowledge models show a subtree of the concept hierarchic and hide distracting details that are not relevant for understating feature implementation (e.g XML tags, Java code). For example, the mappings aggregated to *BeanImplementation* `WeatherServiceImpl` in Figure 3.5 correspond to source code elements spread over two different files – `applicationContext-userservices.xml` and `WeatherServiceImpl.java`. This way, developers are conditioned to directly navigate to source code keeping focus only on the desired concept instances without getting lost. Clearly, a scattered, core-oriented technique does not support this kind of reasoning even when traceability is partially supported by views or general-purpose models. Furthermore, DKMLs maintain mappings to source code elements instantiating concepts, which might be physically located in different places.

Developers can also quickly search for all uses of a certain concept instance. That is, to figure out the impact of feature assignments first they do not need to waste time investigating all the source code since the relevant parts are abstracted as domain knowledge model elements. Moreover, technically, developers can automatically search on domain knowledge models to discover when a feature assignment impact any other concept instance. For example the *Bean* `WeatherService` from Figure 3.5 defines via standard cross-element references that depend on the *Beans* `CityDAO` and `WeatherUserServiceDAO`. Therefore, as we detail in next Chapter, based on such cross references developers can inspect models by searching for all places where a certain model element is referred or having the end of a reference marked when they select an association to inspect. As a direct benefit, even when the configuration knowledge contains complex constraints it still easy for developers to understand the impact of features over concepts instances once this information is clear in domain knowledge models.

DKMLs also encode the knowledge about code customization using concept instances declared into code configuration. From the developers point of view, it becomes particle to observe the impact of features assignment to concept instances since DKMLs avoid the need to jump from code configuration to customization and back again. More detail about how this information is encoded in DKMLs abstract syntax will shown in next Chapter. Another class of feature visualization considers context sensitive instantiation of concept instances. With DKMLs the knowledge about framework programming interface is explicitly represented, which further specifies whether each concept instance

Spring Domain Knowledge Model

```
DKM Spring
    Context : application-context
        Bean : EventDAO
            BeanInterface : EventDAO
            BeanImplementation : EventDAOHibernate
        Bean : EventDAO
            BeanInterface : EventDAO
            BeanImplementation : AcademicEventDAOHibernate
        Bean : EventDAO
            BeanInterface : EventDAO
            BeanImplementation : TravelEventDAOHibernate
```

Figure 3.6: Spring Domain Knowledge Modeling - *Bean* EventDAO.

will be available or not. Therefore, DKMLs make it possible to developers know whether a feature assignment is missing by accident or on purpose.

Now, developers do not need to examine in detail unnecessary code or even reading the framework documentation in order to discover the full range of concepts and their intrinsic instantiation rules, which in most of cases is cumbersome and expensive. Furthermore, engineering product lines with DKMLs also ensure that developer will not forget to observe an important part of source code. It is essentially important because programmers tend to spend more time understanding the code than working with it. A quicker recognition of features tends to significantly improve the speed of software product line comprehension. Consequently, the discussed benefits result in less development effort and chances of introducing errors.

It is also worth to note that the domain knowledge models emulate some sort of modularity. A developer can trace a feature or feature expression from the feature model to its implementation summarized in a specific subtree. Nevertheless, selecting the necessary context information is an important design decision when visualizing feature code. Without context, developers cannot properly understand the abstractions or source code elements in isolation. For example, in Figure 3.6, the Spring-DKM encompasses tree *Beans* called `EventDAO`, where each one is associated with a different *BeanImplementation* model element (e.g., EventDAO, AcademicEventDAO, TravelEvent). A view of configuration knowledge just relating *Beans* and features would not be helpful in understanding the feature code. As the context information is insufficient, developers have to look into other models or at least the source code to understand which concept instance actually is implementing the feature. Therefore, we choose to leave the whole structure (e.g., instances of the parent concepts) that affects the feature code as context. As additional information, we also leave code mappings to directly indicate the physical source code elements implementing features.

However, when the context is too large, the benefit of a feature code visualization technique might become irrelevant. Therefore, it is also possible, as we show in next Chapter, to provide a visualization on single or multiple features. Visualization on feature (Kästner et al. 2008) includes only concept instances assigned to of a given feature. For example, X and Y are included for the Feature X, but not the K and W, since it does not contain any feature code, or not the U and A that are assigned to the Feature K.

Concluding, with models representing the knowledge about domain-specific concepts, we solve the problems outlined in Chapter 2. First it improves feature code visualization by putting in evidence which source code elements belong to a feature as framework concepts, at the same time that ensure code configuration and customization assigned to the same feature. Moreover, as DKMLs results in structured models, they also facilitate the navigability through association between concept instances – both parent-child and references – in addition to support the implementation of more advanced functionalists such as mark occurrences of model elements. DKMLs also benefit developers with support for visualizing the impact assigning feature to associated concept instances. Finally, as DKMLs encodes the framework's programming interface and supports the specification of additional constraints, it also support developers reasoning about concept configuration overloading in scenarios in which the constraints that enforce concept instantiation are context sensitive. More details about the mentioned benefits will be presented next.

Visualizing on source code Domain knowledge model is not exactly the appropriated notation that developers can rely on for visualizing fine-grained variability. Intentionally, they are not designed to represent assignments of features to source code elements that are not instantiating abstractions. Indeed, we observed that framework-based software product lines not only implement features as concept instances but also present fine-grained variability that are directly related to source code statements (see Chapter 2). Moreover, feature assignment on source code might benefit developers with further support for keeping track of the according feature belonging to the annotated code element when performing some maintenance tasks. Visualizing on source code can show how multiple features interact in the same file much more directly.

To overcome this, we decide to provide support for visualizing features directly on source code. The idea is to virtually project both fine-grained and coarse-grained variability in the source code realizing them. We use the term project because actually annotations are not physically stored in the source, but derived from assignment of features to model elements. There are two ways in which an annotation is created:

1. if the investigated file contains fragments assigned to features in the implementation model, an annotation is created. Note that creating such annotations is trivial because elements of the type fragment maintain a direct reference to source code elements. Therefore, we just annotate the specified location in the source code with the associated feature.

2. if the investigated file contains fragments aggregated to domain knowledge model element assigned to features, an annotation is also created. However, projecting features assigned to domain model elements on source code might poses some challenges. Especially for nested elements in the hierarchic, in some cases, it might be compelling to annotate their related code fragments with features assigned to some of their parent. For example, observe in Figure 3.5 that assigning the *Bean* `WeatherService` to a feature does not guarantee that an annotation will be created to *Fragment* `<bean id="WeatherService" class="...WeatherServiceImpl>"` aggregated to its *BeanImplementation* `WeatherServiceImpl`. Observe that in other cases it might pollute the source code, for example annotating every nested element with its parents feature. For example, it does not make sense annotating every *Bean* declared inside a *Context* file with the feature assigned to it. In such case, instead of improve feature code visualization it might complicate developers on properly comprehend the source code.

We implemented one solution for the second case, however the presented case remains open to explore. In our implementation, we search for two different patterns. First, we recursively search in the parent hierarchy for the first occurrence of a model element assigned to a feature, and that is not associated to any element from the implementation model. We assume that children of supporting elements are their actual implementation. For example, this pattern is enough to guarantee that every *Fragment* aggregated to *BeanImplementation* will be annotated with its parent feature, in case, the feature assigned to instances of *Bean*. Second, we recursively search in the parent hierarchic for occurrences of model elements assigned to features until that element is associated to an implementation model element that is not container of the current *Fragment*. Usually, the *Fragment* represents the code customization of related concept instance. For example, suppose the developer assigning the *ConstructionInjection* `CityDAO` to a feature X. In such case, the *Fragment* `CityDAO cityDAO` will be annotated with the feature X, which make sense.

3.4 Consistency Checking and Guidance

Checking consistency of framework-based product lines early, that is, during configuration knowledge editing time, is one of the primary applications of DKMLs. Consistency checking is essential because errors on product lines may appear only in some products with specific composition of features (Kästner and Apel 2008). Due to the exponential number of products in software product lines, deriving, separately compiling every variant in isolation is usually infeasible. Even worse, the types of errors that we observe in framework-based software product lines can only be detected at running time (see Chapter 2), that is, at framework interpretation time. Therefore, some errors can still undetected until the product perform a certain execution path.

In this section we address first structural errors (Section 3.4.1). As stated in Section 3.4.1, structural errors occur when a product is ill-formed regarding the right ways of decomposing framework-provided concepts. For example, in Jadex, *Plans* represent the agent's means to act in its environment. *Agents* in general act in response to occurring *Events* or *Goals*. To indicate in what cases a *Plan* is applicable the *Trigger* concept is used (see Figure 2.2). So, every *Plan* instance must define a *Trigger*. In this case, developers cannot assign *Triggers* to optional/alternative features, once those might cause a structural error when removed. After that, we propose a certain level of guidance for ensuring that developers always follow the stipulated framework's instantiation constraints when creating the domain knowledge models (Section 3.4.2). For example, prevents that *Bean* instances define their code configuration but not the code customization. Finally, we use a constraint satisfaction programming formulation of the configuration knowledge to ensure that every feature assignment are respecting both feature model and DKMLs semantics.

3.4.1 Disciplined Feature Assignment

Disciplined feature assignment is the most basic solution that prevents errors in software product lines (Kästner et al. 2009). In case, they are assignments to domain knowledge model elements that do not introduce structural errors when those model elements are deleted. As discussed in Chapter 2, the code-oriented techniques are prone to structural errors. They only support arbitrary assignment of source code elements to features and checking them taking into account the framework's programming interface is not possible.

The reason that code-oriented techniques fail to prevent structural errors is that they only consider the source code structure. Thus we still able to remove any element from the source, since it does not introduce any syntax or type error (Kästner and Apel 2008, Kästner et al. 2009, Czarnecki and Pietroszek 2006). Even when they support expressing high-level constraints, those are general-propose and still not considering the right way of configuring the source code according to framework's programming interface.

As DKMLs encode the structure defined by framework's programming interfaces, they specify which concepts may appear, where and how they are related. But it needs an insight into which assignments are disciplined and which are not. To determine this we rely on the `selectable` abstraction, as presented in Section 3.2. From this information, we can infer which elements are mandatory – not defined as `selectable` – and optional in the DMKL abstract syntax. Instances of `selectable` elements are those which can be assigned to features safely. For example, if we allow only assignments of features to *Plans* in Jadex we guarantee that removing the related code fragment will not introduce any syntax error regarding the Jadex programming interface. Instead, developers cannot assign the *Trigger* concept defined inside a *Plan* code configuration. As mentioned, according to Jadex instantiation constraints every *Plan* must expresses an *Trigger*.

Therefore, initially two configuration rules provide the foundational mechanisms for guaranteeing consistency, as also stated in (Kästner et al. 2009):

- **Optional rule**. Only model elements defined as optional according to framework's instantiation constraints are allowed to be associated to features and consequently removed from the common code.
- **Hierarchical rule**. This kind of rule propagates the feature of the parent to its children. For example, if an *Bean* is removed, all children are removed as well.

It is worth to note that disciplined assignments can only guarantee correct assignments for already correct DKMs and does consider references between model elements. Next, we present how DKMLs might guide developers on the correct construction of DKMs and after that they are used to check the consistency of the entire product line.

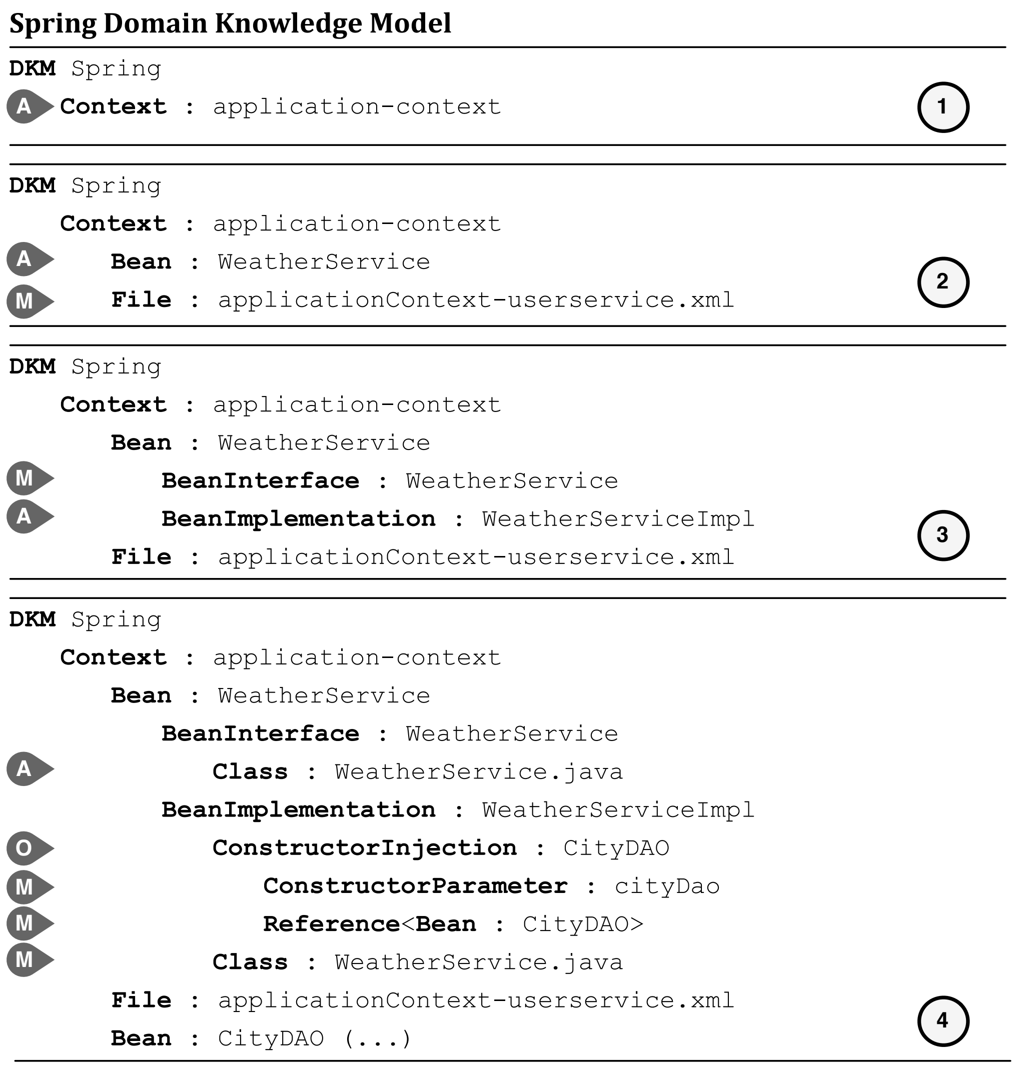
3.4.2 Guidance

Only preventing errors regarding the right ways in which framework-provided concepts may be decomposed is not sufficient, specially because in order to analyze the DKMLs structure and to reason about feature assignment, already the models must not contain structural errors. However, developers often are not aware of the entire set of framework-provided concepts and instantiation constraints, and the creation of domain models is also subject to errors in general.

Therefore, in addition to be highly desirable the storage of information about the proper way of instantiating concepts in DKMLs abstract syntax, it is also fundamental to use this information for guiding inexperienced developers during configuration knowledge specification. In this context, we encode a guidance model as editing operations. The concept of edition operations has already suggested as a facility to guide developer toward a consistent model (Xiong et al. 2009). The idea is that regardless its abstract syntax, models can be represented as a sequence of operations performed to construct it rather than by the set of elements it contains. We only consider one kind of inconsistency: structural. Structural consistency constraints, in this case, define the existence of elements and association that should hold between them. These rules can be compared with well-formedness rules and syntactic rules.

One instance of a DKML is a domain knowledge model that mimics the instantiated framework-provided concepts on the source code, and captures the set of all possible configurations of the software product line. These elements must follow the rules given in the DKML abstract syntax. Basically, the `classes` in the abstract syntax limit the possible elements, and `references` limit the possible association between model elements. The cardinality and other constraints again limit the number of possible elements or impose the creation of new ones. The validity of models, therefore, can be analyzed by considering the validity of each element and the existence or absence of association to other elements. The lower bound in the cardinality determines whether there must be an element at the end of a `reference` or not. If the lower bound is zero, the existence of `reference` is optional, otherwise it is mandatory. The upper bound determines the maximum number of certain kind of elements in one `reference`. Considering this information, simple editing operations are offered by computing the semantics of DKMLs abstract syntax, in accordance with the following rules:

– (ForAll) - `<1-*>` - The concept instance C_j can incorporate at least one reference for each type of concept C_i in the set $\{C_1...C_n\}$.



Spring Domain Knowledge Model

```
  DKM Spring
A Context : application-context                                   1

  DKM Spring
    Context : application-context
A     Bean : WeatherService                                       2
M     File : applicationContext-userservice.xml

  DKM Spring
    Context : application-context
      Bean : WeatherService
M       BeanInterface : WeatherService
A       BeanImplementation : WeatherServiceImpl                   3
      File : applicationContext-userservice.xml

  DKM Spring
    Context : application-context
      Bean : WeatherService
        BeanInterface : WeatherService
A         Class : WeatherService.java
        BeanImplementation : WeatherServiceImpl
O         ConstructorInjection : CityDAO
M           ConstructorParameter : cityDao
M           Reference<Bean : CityDAO>
M         Class : WeatherService.java
      File : applicationContext-userservice.xml
      Bean : CityDAO (...)                                        4
```

Figure 3.7: Detailed creation of Spring-DKM following the guidance rules.

- (Mandatory) - `<1-1>` - The concept instance C_j must incorporate one reference for each type of concept C_i in the set $\{C_1...C_n\}$.
- (Optional) - `<0-*>` - The concept instance C_j can incorporate zero or more references for each type of concept C_i in the set $\{C_1...C_n\}$.

For example, supposing the creation of Spring-DKM as exemplified in Figure 3.7. At the first moment, a developer creates the *Context application-Context*. As the association of *Context* and *Bean* has a cardinality `<1-*>`, the developer is informed about the need of creating a instance of one *Bean*, in this case suppose that the developer decided to create the *Bean* `WeatherService`. Once the `WeatherService` is created, the developer is asked for creating one and only one instance of the *BeanInterface*. Observe that she is not allowed to create more than one instance of such concept. At the same time, she is also asked for creating at least one instance of the *BeanImplementation*. In this case, she created the `WeatherServceImpl`. Instance of *BeanImplementation*, as defined in Spring documentation, can provide one, but only one, type of dependency injection. Note that it is not an essential concept. In the case of the `WeatherService`, it was defined two *ConstructionInjection* to *Beans*, `WeatherUserService` and `CityDAO`. Observing only the `WeatherUserService` we can see that the developer must define: (i) to which existing *Bean* the `WeatherUserService` refers to – `WeatherUserServiceDAO`; (ii) one instance of *ConstructorParameter* – `WeatherUserService`.

Moreover, as all mappings to source code are also encoded in the abstract syntax, the developers must follow the same rules. In this case, for example, when they create the *BeanInterface* `WeatherService` they are notified about the need of defining a reference to one class of the implementation model due to the `<1-1>` cardinality of such association. Therefore, the steps that developers must follow when defining concept instances is clearly defined in the DKMLs abstract syntax and can be checked for consistency.

Therefore, based on this previous scenario, we defined two more configuration rules that complement the foundational mechanisms for guaranteeing consistency:

– **Realization rule**. When a model element is removed, all associated source code elements must be removed as well. For example, when an *BeanImplementation* is removed also the XML fragment declaring its existence and Java class implementing its behavior must be removed as well.

– **Referential rule**. A referential rule constrains the presence of two elements. For example, if a instance of a *Bean* concept is chosen to be removed from the common code, all related *Beans* instances must be removed as well.

3.4.3 Consistency Checking

In second part of the analysis, we use a global verification to check consistency of feature assignment to both spaces: domain knowledge and source code. Here is sufficient for each assignment to check whether the they are satisfied together. We have opted for a consistency approach based on Constraint Satisfaction Problem (CSP) (Tsang 1993) since it affords us with great expressiveness and feedback. We can express local and global well-formedness constraints in a uniform way, which allows constraint spanning multiple languages (e.g., feature model, configuration models, DKMs). Feedback messages also can be derived from counter example representing a specific product that will not work properly. The feedback message can be presented to the developer for further debugging.

Constraint Model Definition In order to define the constraint satisfaction programming model used to automatically infer a configuration of the source code from a feature model configuration, we begin by precisely defining the entities that compose the models and the relationships between elements from feature, DKMLs and implementation models. After that, we describe an interpretation of these entities and relationships as variables and constraints, respectively, of a constraint satisfaction problem.

The following definitions establish the building blocks of the constraint satisfaction problem.

- ***Feature Model***. Feature model has a diversity of forms. They can have attributes, cardinalities and various types of constraints (e.g., includes, excludes). We abstract from these different representations and, instead, directly operate on the basic semantics. The feature model is assumed as a set of features F.
- ***Implementation Model***. The implementation model is a set of model elements E_I.
- ***Domain Knowledge Model***. One domain knowledge model is defined as a set of model elements E_A and a set of relations $R_A \subseteq E_A \times E_A$ and a set of relations $R_I \subseteq E_A \times E_I$. Relations $r_A \in R_A$ are distinguished between referential and hierarchical. Relations $r_I \in R_I$ are assumed as realization relations.
- ***Configuration Model***. The configuration model is a set of relations $R_C \subseteq F_E \times E_A \cup E_I$, where F_E is the set of all valid boolean expressions b that can be built from the set of F.

Variables The model of features represents the actual configuration of the product line. It encompasses a set of variables V where each variable $v \in V$ represents a feature $f \in F$. The domain of each variable v is the same, 0 or 1 (i.e., false or true, respectively). It means that, when v represents a selected feature, its domain value is defined as 1 (i.e., true); on the other hand its domain value is defined as 0 (i.e., false). The constraint solver finds an evaluation (i.e., *instance*) of other two sets of variables A and I based on the values defined in the model of features.

The set A represents the elements from E_A. The domain of each variable $a \in A$ is defined as 0 or 1. At the end of the constraint problem evaluation, if a_{e_A} represents an element e_A that must be part of the derived domain knowledge model, its value is set to 1. Otherwise, if m_{e_A} represents an element that must be removed from this model configuration, its value is set to 0. Finally, the variables $i \in I$ represent elements from the set E_I and in the same way they assume 0 or 1. Similarly, if i_{e_I} represents an element e_I that must be present in the product configuration, its domain value is defined as 1, if not, its domain value is defined as 0.

Constraints Each assignment from feature to solution space element is a bi-implication constraint constructed from the set of relation R_C, where the positive evaluation of the boolean feature expression b implies in the derivation of the associated *Selectable* element $e \in E_A \cup E_I$ – and vice-versa. Observe that elements $e \in E_A \cup E_I$ are represented in the constraint programming model as variables $w \in M \cup A$.

$$(b = 1) \Leftrightarrow (w = 1)$$

The relations between elements from the solution space models become constraints among variables $a \in A$ and $i \in I$. We distinguish three kinds of constraints:

– *Realization.* Each relation from domain knowledge models to implementation model is interpreted as a bi-implication constraint, where the derivation of the element $e_A \in E_A$ implies in the derivation of the element $e_I \in E_I$ – and vice-versa.

$$(m_{e_A} = 1) \Leftrightarrow (a_{e_I} = 1)$$

– *Hierarchical.* Each hierarchical relation is an implication constraint, where the derivation of the child element $c_A \in E_A$ implies the positive derivation of the parent element $p_A \in E_A$.

$$(m_{c_A} = 1) \Rightarrow (m_{p_A} = 1)$$

– *Referential.* Each referential relation is an implication constraint, where the derivation of the element $e_A \in E_A$ implies the positive derivation of the referred element $r_A \in E_A$.

$$(m_{e_A} = 1) \Rightarrow (m_{r_A} = 1)$$

Constraint Satisfaction Problem Solving and Product Derivation To solve the constraint satisfaction problem, the constraint solver is invoked utilizing its propagation functionality. The goal is to find an assignment to all variables in A and I that do not violate any constraint. After the evaluation of the constraint problem, the values of all variables in A and I serve as input to decide which source code elements must be selected/removed to derive the product.

The constraint programming model also describes the set of all valid composition of source code elements. Since it is described as constraints over the variables in F, A and I, a valid evaluation of these variables always yields a valid product regarding the rules defined in the domain knowledge models. However, when any constraint violation is found we can assume the configuration knowledge as inconsistent and that a derivation, which conforms with the feature selection and respect all constraints imposed by the configuration model and domain knowledge models, does not exist. In this case, the developer needs to revisit the configuration knowledge specification.

3.5 Summary

In this Chapter we presented the key ideas behind engineering framework-based software product lines with domain-knowledge modeling languages. DKMLs express framework's programming interface and can be used for defining models that describe how features are assigned to concept instances in the scope of an DKML. We showed the definitions and properties for modeling framework's programming interface using DMKLs. We presented how DKMLs improve feature code visualization by putting in evidence which source code elements belong to a feature as framework concepts. At the same time, we maintain all benefits of traditional code-oriented techniques. Our technique can express fine-grained variability, provides easy to adopt modeling language, is uniform for many languages artifacts and frameworks. Finally, we presented how framework's programming interfaces encoded in DKMLs support consistency checking and guided configuration knowledge specification. In next Chapter, we present the realization of such presented ideas in a tool called GenArch$^+$.