

Product Line Engineering with UML¹

Tewfik Ziadi, Jean-Marc Jézéquel, and Frédéric Fondement
IRISA, Campus Universitaire de Beaulieu, 35042 Rennes Cedex, France
{Tewfik.Ziadi, Jean-Marc.Jezequel, frederic.fondement}@irisa.fr

Abstract

The Unified Modeling Language (UML) is a standard language for the object-oriented analysis and design. We propose in this paper an approach based on the UML that supports the Product Line Engineering. We provide a set of patterns for modeling variability issues of a Product Line Architecture, we define architectural constraints for Product Line expressed in UML as meta-level OCL constraints, and we propose a method based on the use of a creational design pattern to automate the derivation process. This makes it possible to automatically derive a given product from the set of all possible ones, and to specialize its model accordingly.

1. Introduction

Software Product Line (SPL) captures "commonality" and "variability" between a set of software products in the same domain. Commonality designates elements that are common to all products while variability designates elements that may vary from a product to another one. Software Product Line engineering aims at improving productivity and decrease realization times by gathering the analysis, design and implementation activities of a family of systems. It is based on the reuse of assets instead of working from scratch. A Software Product Line Architecture also called a reference architecture is a generic architecture from which the model of each product can be derived. The role of software product line architecture is to describe commonalities and variabilities of the products contained in the Product Line (PL) and, as such, to provide a common overall structure.

To model SPL with the UML (Unified Modeling Language) [21], we need mechanisms to specify variabilities and commonalities, and techniques to derive products. We also need to manage a set of constraints that specify variation point dependencies in the PL.

This work focuses on the PL engineering with the UML. It provides a set of patterns for modeling variability issues of a Product Line Architecture, and it proposes an approach based on a creational design pattern to derive product models from a PL architecture modeled by the UML. The derivation process should preserve PL coherence, so we have defined and specified a set of PL constraints as OCL (Object Constraint Language) meta-model constraints. To illustrate our approach, we use a Mercure PL.

The paper is organized as follows: Section 2 briefly presents the Software Product Line Engineering approach and its concepts, In section 3, we present some mechanisms to specify variability in the UML, and the section 4 presents the Mercure PL. Section 5 presents PL constraints and their specification with the OCL, and the section 6 illustrates the derivation process. Section 7 presents mechanisms to specify variability in UML sequence diagrams. Finally section 8 concludes this work.

¹ This work has been partially supported by the CAFE European project. Eureka Σ! 2023 Programme, ITEA project ip 0004

2. The Product Line Engineering

2.1. The approach

The general process of Product Line Engineering, as found in the literature [4,13,20], is illustrated in the Figure 1. We distinguish two main activities:

Domain Engineering. The domain engineering activity is twofold:

- Collecting, organizing, and storing past experiences in building systems in the form of reusable assets (i.e. reusable work products) in a particular domain,
- providing an adequate means for reusing these assets when building new systems [13].

The term *Developing for reuse* is often used to characterize the Domain Engineering. It can be divided in three main processes: *Domain Analysis*, *Domain Design*, and *Domain Implementation*. The domain analysis consists in capturing information and organizing it as a model. Some methods, such as FODA (Feature-Oriented Domain Analysis) [14] propose a set of notations for the domain modeling using the notion of "features" to refer to products properties. The domain design consists in establishing the product line architecture. The domain implementation consists in implementing the architecture defined during the domain design as software components.

Application Engineering. The application engineering activity consists in building systems based on the results of Domain Engineering. During application requirements of a new system, we select the requirements from the existing domain model, which matches the customer's needs. We assemble applications from the existing reusable components. The term *Developing by reuse* is used to characterize the application engineering activity.

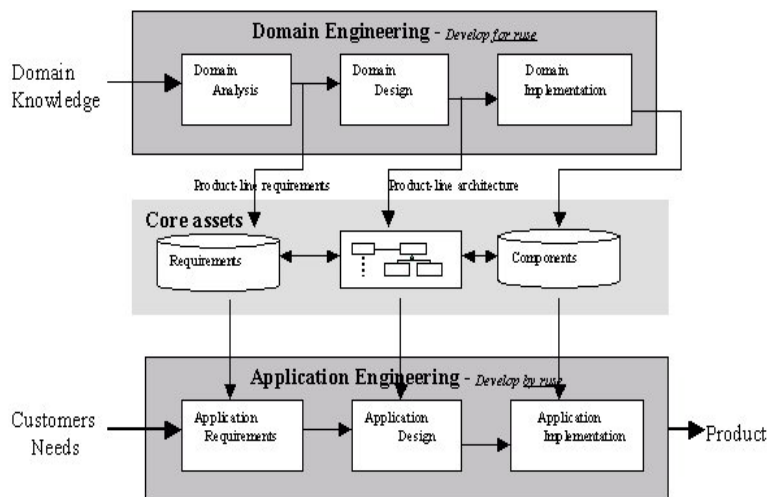


Figure 1. The general process for Product Lines Engineering

2.2. Variability

Variabilities are characteristics that may vary from product to another. The main challenge in the context of software Product Lines approach is to model and implement these variabilities. Even if the product line approach is a new paradigm, managing variability in software systems is not a new problem and some design and programming techniques allows us to handle variability (see the related work section); however outside the Product Line context, the variability concerns a *Single Product*, i.e variability is inherent part of the a single software and it is resolved *after* the product is delivered to customers and loaded into its final execution environment.

In the product line context, variability should explicitly be specified and it is a part of the product line and, in contrast with the single product variability, PL variability is resolved *before* the software product is delivered to customers. [2] calls the variability included in the single product "the run time variability", and the PL variability is called "the development time variability".

3. Modeling Variability with the UML

The Unified Modeling Language (UML) [21] is a standard language for the object-oriented analysis and design. It defines a set of notations (gathered in diagrams) to describe different aspects of the system: use cases, sequence diagrams, class diagrams, component diagrams and statecharts are examples of these notations. The UML also defines stereotypes, tagged values and constraints as extension mechanisms that can be used to extend the UML meta-model. In this section, we try to show, through an ad-hoc example, how variability can be modeled using UML notations (we only use here the UML class diagram; in the section 7, the variability in UML dynamic diagram is presented). The example concerns a digital camera. A digital camera comports an interface, a memory, a sensor, a display and may comport a compressor. The main variability in this example concerns the presence of the compressor and the format of images, which can be parameterized. As previously stated, variability can occur in the single product as well in the Product Line. Firstly, the camera system is modeled as a single product to show how UML mechanisms can be used to model the *Single Product Variability*, and secondly, we use the same example but as a PL to illustrate the modeling of the *PL Variability* with UML.

1. *The Single Product Variability.* If we consider a camera system as a single product, i.e: the same camera product will be delivered to all customers. Figure.2. shows its class diagram. A camera aggregates one interface, one sensor, one display, and one memory and it may aggregate a compressor. The variability in this diagram is modeled using two mechanisms:
 - a. The optionality of the compressor is specified by the cardinality 0..1 between the camera and the compressor. This cardinality specifies that cameras can have zero compressor,
 - b. the parameterization of the memory is modeled by the UML class template with one parameter that specifies the type of images which can be supported.

The UML cardinality constraints and class templates allow us to specify the variability in the single product; this variability is resolved at the run time. Indeed, the camera system used by all customers include the compressor class and only at the run time that the variability specified by the 0..1 cardinality will be handled. In the same, only at the class instantiation that the memory parameter will be resolved with the specific type. Moreover the cardinality constraint and the parameterization, UML includes mechanisms inherited from the *object oriented paradigm* such as: the generalization and the specialization that can model the variability in a single product. They are used to model the commonalities between the variants of a variation point in an abstract class (or interface), and expressing the differences in concrete subclasses (each variant implements the interface in its own way).

2. *The Product Line Variability.* The Product Line variability should be explicitly specified. In the camera example, it may be desirable to have two separately camera products: the first one with no support for the compression, and the second comports a compressor. Defining the camera as a Product Line needs to explicitly specify that the compression is an optional feature. Using UML, the model of the camera PL should be as a reference model from it each product is derived and created. Figure.3. shows the UML class diagram of the camera PL, the stereotype <<optional>> associated to the compressor class specifies that the compressor is an optional class, i.e it can be omitted in some products models. UML extension mechanisms, in particular stereotypes, can be used to specify the PL variability.. The variability specified by stereotypes is resolved when the product model is derived. For example, the UML class diagram of the camera product that does not support the compression is obtained by deleting the compression class from the camera PL class diagram. Notice that the UML class diagram of the camera PL also includes the memory class template; however this variability is not considered as PL variability because the both camera products (with and without the compressor) include it, and it is resolved at the run time.

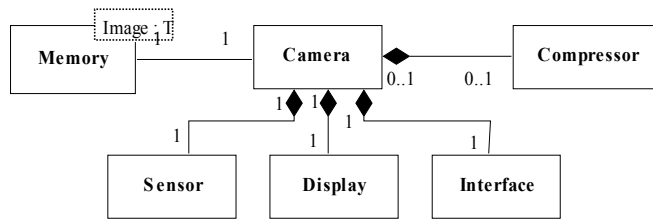


Figure.2. The UML multiplicity and the class template to model variability in a single product

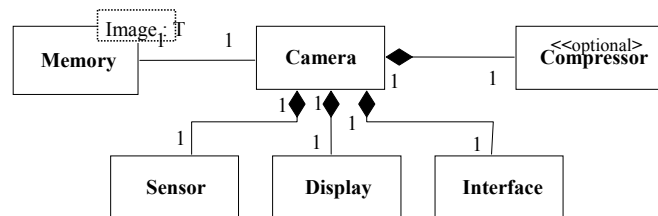


Figure.3. UML stereotypes to model variability in the PL

4. The Mercure Case Study

As a case study for evaluating our approach, we consider the Mercure PL, which is a family of SMDS (Switched Multi-Megabits Data Service) servers whose design and implementation have been described in [10,11]. It can abstractly be described as a communication software delivering, forwarding, and relaying “messages” from and to a set of network interfaces connected into an heterogeneous distributed system. The Mercure PL must handle variants for five variation points: any number of specialized processors (Engines), network interface boards (NetDriver), levels of functionality (Manager), user interface (GUI) and support for languages (Language)

4.1. The Variability in the Mercure PL

To identify variabilities in the Mercure PL, we specify its domain model using FODA notations, slightly modified and extended by [13]. We use a set of feature kinds to specify variability and commonality:

- *Mandatory features*: to specify features that are common to all products, we use mandatory features whose ancestors are also mandatory. Mandatory features are shown in the FODA diagram by nodes with black circles.
- *Optional features*: it represents features that can be omitted in some products; it is shown by nodes with an empty circle.
- *Or-features*: a feature may have one or more sets of direct or-features. If the parent of a set of or-features is included in the description of a specific product, then any nonempty subset from the set of or-features is included. The nodes of a set of or-features are pointed to by edges connected by a filled arc.

Figure 4. shows a feature diagram of the Mercure PL. The Mercure consists of Engine, Net Driver, Manager, GUI, and Language; all these features are mandatory. The Mercure product may support one or more of Engine 1,..Engine N, we use FODA or-features to represent it. In the same way, we define all NetDrivers and Managers dimensions. However all

Mercure products should support one GUI, which is GUI 1, so it is defined mandatory. Other GUIs are defined as FODA or-features. We distinguish two categories of languages: Language Cat1 and Language Cat2, all products should support the Language 1.1 of the first category; Language 2.1, and Language 2.2 of the second category are optional.

The FODA notations allow us to specify dependencies relationships, called “composition rules”, between domain features. FODA supports two types of composition rules: the *requires rule* that expresses the presence implication of two features, and the *mutually-exclusive rule* that captures the mutual exclusion constraint on feature combinations. Two rules are identified in the context of the Mercure PL: a *requires rule* is added between the Engine 1 and the Net Driver 1 while a *mutual-exclusion rule* is added to specify that GUI 1 do not supports Language 2.2 (see Figure 4).

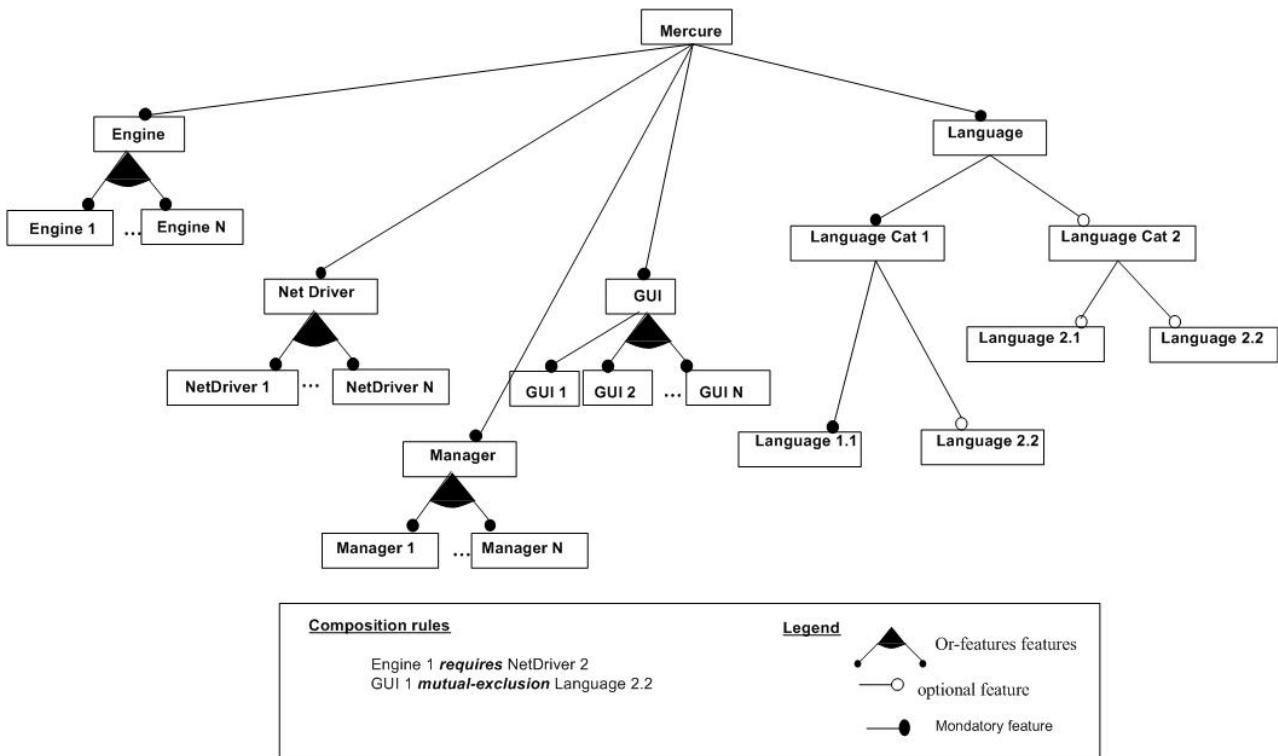


Figure 4. The FODA diagram for the Mercure PL

4.2. The UML class diagram of the Mercure PL

The UML class diagram is used to specify the architecture of the system in term of classes and their relationships. To describe the Mercure PL architecture, we use the UML class diagrams. As previously specified in the FODA diagram of the Mercure PL, the Mercure product may support a set of Engine among Engine1, Engine2,... EngineN. Using UML, we define an abstract class called Engine and the several dimensions as subclasses; in the same way we specify other variation points (NetDriver, Manager, GUI, and Language). The stereotype <<optional>> is used to show the optionality; this stereotype can be applied in the class diagram to classes, interfaces, and packages.

Figure.5. shows the UML class diagram of the Mercure PL. It basically says that a Mercure system is an instance of the MERCURE class, aggregating an ENGINE (that encapsulates the work that Mercure has to do on a particular processor of the target distributed system), a collection of NETDRIVERS, a collection of MANAGERS (that represent the range of functionalities available), and the GUI that encapsulates the user preference variability factor. A GUI has itself a collection of supported languages, which are classified into two categories.

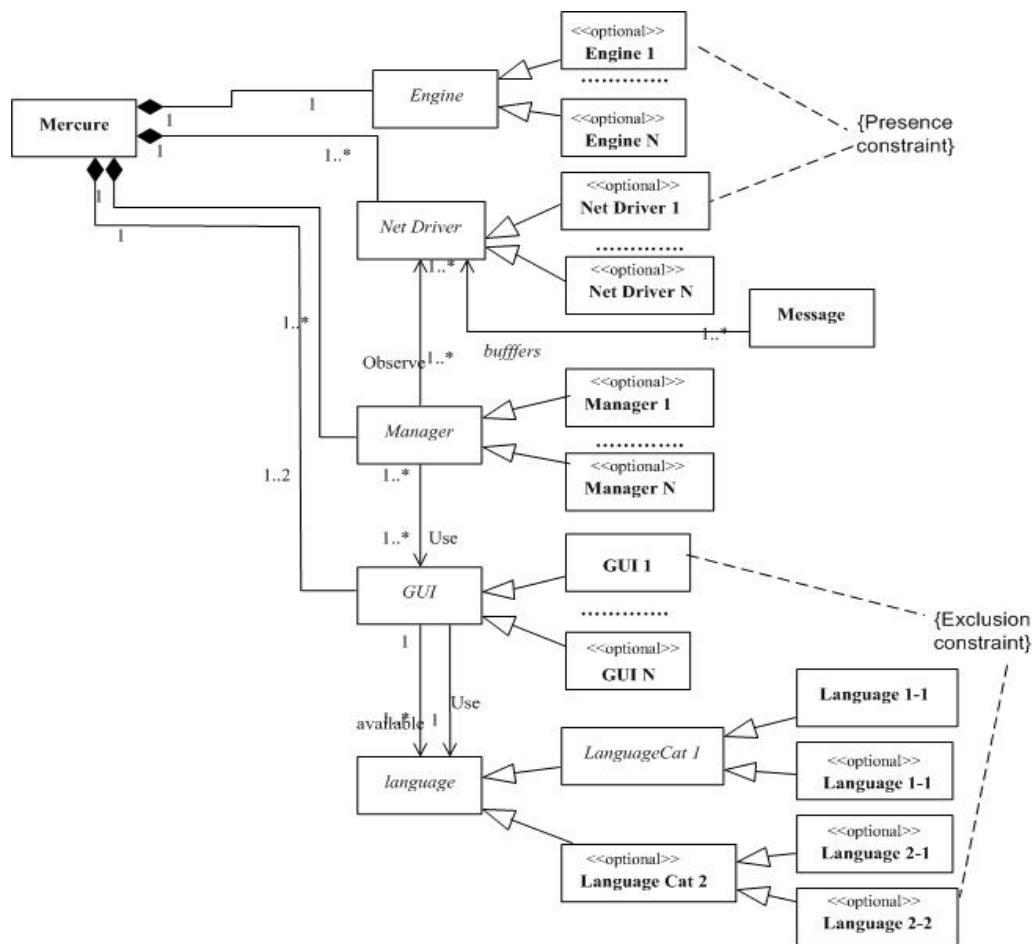


Figure 5. The Mercure Product Line UML class diagram

A UML class model of a specific derived product of Mercure can include an optional number of Engines, Network Drivers, Managers, GUIs, and Languages; so these features are defined as abstract classes, and we specify variants as concrete subclasses with the optional stereotype. All Mercure products should at least support one mandatory language (LANGUAGE1-1), and one GUI (GUI1), so these subclasses are defined without the optional stereotype.

Defining variation points as abstract classes and each possible variant as subclass with the optional stereotype is what we call the “abstraction variability pattern”.

5. Managing the PL constraints

[15] considers that constraints are parts of PL architectures. Constraints define coherence rules and relationships between elements in the architecture. As shown previously, FODA composition rules allow us to specify relationships between domain features. Using UML, some work such as [17] use UML stereotypes to show dependencies between classes.

The Object Constraints Language (OCL) [9] allows us to attach constraints to UML models. These constraints can be defined at meta-model level as well as model level. In the context of Product Lines, we have identified two types of constraints: *generic constraints* applying to any PL, and *specific constraints* associated to a specific Product Line and we propose to define them as OCL meta-model constraints.

5.1 The Generic Constraints

The introduction of variability, especially the optionality (specified by the <<optional>>stereotype), in the PL model allows us to improve genericity but it can generate some incoherence. For example, if a non-optional element depends on an optional one, we risk deriving an incomplete product model. So the first type of product line constraints defines structural properties of any product line model to preserve its coherence. UML can be extended by defining a set of stereotypes and a set of meta-level constraints that are often related to these stereotypes. So the idea for defining generic constraints is to associate a set of constraints to the relevant stereotypes, this solution was already used in [6] to define design pattern occurrences in the UML. These constraints are represented as OCL meta-model level constraints and they will be evaluated on any product line model, see Figure 6. The generic constraints may be seen as *well-formedness rules for the UML modeled product lines*.

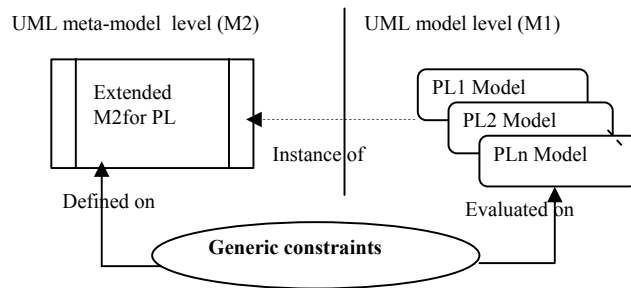


Figure 6. Generic constraints as OCL meta-level constraints

Examples of the generic constraints

Generic constraints aim to preserve the PL model coherence. In the case of the static model represented by the UML class diagram, we have defined the dependency and the inheritance constraints:

The dependency constraint. A dependency in the UML specifies a required relationship between two or more elements. It is represented in the UML meta-model [21 pp 2.15] by the meta-class *Dependency* (see appendix), it represents the relationship between a set of suppliers and clients. An example of the UML Dependency is the "Usage", which appears when a package uses another one. If a non-optional element is depending on an optional one, there is incoherence in the model. To specify this rule, we add the following constraint as an invariant to the *Dependency* meta-class in the UML meta-model, where *isStereotyped(S)* is an auxiliary primitive indicating if an element is stereotyped by a string S (see appendix):

```
context Foundation::Core::Dependency
-- For each Dependency: if the supplier is optional the client should be optional too
inv:
  self.supplier->exists(S:ModelElement |
    S.isStereotyped('optional')) implies
  self.client->forall(C:ModelElement |
    C.isStereotyped('optional'))
```

The inheritance constraint. Optional classes in Product Line model can be omitted in some products then, if a non-optional class inherits from an optional one, perhaps there is incoherence in the product model. However, in some cases, in particular when the product line model includes the multiple inheritance, it can be correct. But it is more advisable to generate a warning if the static model includes a non-optional class which inherits from an optional one. The inheritance is represented in the UML by the meta-class *Generalization* [21 p 2.14] (see appendix). The inheritance constraint is added as an invariant to the *Generalization* meta-class:

```

context Foundation::Core::Generalization
-- For each generalization: if the parent is
optional the child should be optional too
inv:
  self.parent.isStereotyped('optional') implies
  self.child.isStereotyped('optional')

```

Applying this to the Mercure PL model, LANGUAGE2-1 and LANGUAGE2-2 classes appear to be defined as optional because their parent (LANGUAGE_CAT2) is optional and there is not a multiple inheritance.

5.2 The Specific Constraints

A fundamental characteristic of product lines is that not all elements are compatible. That is, the selection of one element may disable (or enable) the selection of others. The set of constraints that define variation points dependencies in the specific product line are called “Specific Constraints“. As generic constraints, we propose to specify specific constraints as OCL meta-level constraints. The aim of these constraints is to add dependency relationships between model elements, they are associated to a specific product line and will be evaluated on all products, derived from this PL, see Figure 7. The specific constraints are *parts of the PL model definition*.

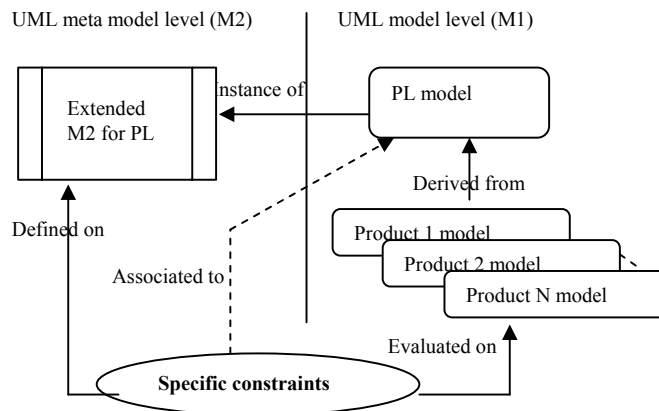


Figure 7. Specific constraints for PL model as OCL meta-level constraints

Examples of specific constraints

A PL class diagram is defined to be as generic as possible and it should include elements related to all products. We have defined the presence and the mutual exclusion constraint as examples of specific constraints and we propose to define them as *Model* meta-class invariants [21 p 2.189]. A *Model* is a namespace that contains a set of *ModelElement* whose names designate a unique element within the namespace.

The presence constraint. This constraint is close to the *requires rule* in FODA, it expresses in a specific PL model that the presence of an optional class requires the presence of another optional class. To specify a require relationship between ENGINE1 and NETDRIVER1 classes in the class diagram of the Mercure PL, we add the following OCL meta-model constraint as a Model meta-class invariant, where the *presenceClass(C)* is an auxiliary operation indicating if a specific class called C is present in the namespace (see appendix):

```
context Model_Management::Model
--The presence in the model of the class called 'ENGINE1' requires the presence in the same model of
the class called 'NETDRIVER2'
inv:
self.presenceClass('ENGINE1') implies
self.presenceClass('NETDRIVER1')
```

The mutual exclusion constraint. This constraint expresses in a specific PL model that two optional classes cannot be present in the same product. As shown previously, GUI1 does not support LANGUAGE_CAT2.2, so the mutual exclusion constraint between their associated UML classes is added as an invariant to the Model meta-class:

```
context Model_Management::Model
-- A class called GUI1 and a class called
LANGUAGE_CAT2 cannot be present in the same model
inv:
(self.presenceClass('GUI1') implies not self.presenceClass('LANGUAGE_CAT2.2'))and
(self.presenceClass('LANGUAGE_CAT2.2') implies not self.presenceClass('GUI1'))
```

In the UML class diagram (see Figure 5.), we use graphical shorthands to show the above constraints.

6. From the Product Line to Products

Once we have analyzed the Product Line and produced the corresponding UML Model, enriched with constraints, we still need to handle the various derivations of products. The PL derivation consists in generating from the PL model the UML class diagram of each product. As shown previously, the PL model is defined by a set of variation points and to derive a specific product model, some decisions (or choices) associated to these variation points are needed. For example, each Mercure product model should choice among the presence or non-presence of all optional classes. So another challenge in the context of PL engineering is to specify a “decision model”.

A decision model represents the set of relevant decisions and their impacts that are needed to identify one single product of the product line [4]. In this section, we propose to use the design pattern *abstract factory* as a model decision and we propose an algorithm for the product model derivation. To illustrate the derivation process, we have defined three products of the Mercure PL:

- **FullMercure**: it is the product that includes all optional elements. Thus, all combinations can be dynamically bound.
- **CustomMercure**: it is a restricted product that supports only two different network drivers (NETDRIVER1 and NETDRIVER2), two languages (LANGUAGE 1-1, which is mandatory and LANGUAGE 2-1) and two GUIs (GUI1, GUI2).
- **MiniMercure**: is a lightest product that supports only ENGINE1, GUI1, LANGUAGE 1-1, MANAGER1, and NETDRIVER1.

6.1. The decision model

In [12], the creational design pattern *abstract factory* [3] is used to refine the several variation points. Our aim in this section is to use this pattern as a design of the PL decision model. The decision model of the Mercure PL is illustrated in the Figure 8. We use an *Abstract Factory*, called *Mercure_Factory* to define an interface for creating variants of

Mercure's five variation points. The class `Mercure_Factory` features one *Factory Method* for each our 5 variation point (`new_gui()`, `new_language()`, `new_network_manager()`, `new_netdriver()`, and `new_engine()`). These Factory Methods are abstractly defined in the class `Mercure_Factory`, and given concrete implementation in its subclasses (e.g `FullMercure`, `CustomMercure`, and `MiniMercure`), called *concrete factory*. Each concrete factory concerns one Mercure product. We use stereotypes to restrict the returned type of Factory Methods to the possible one. For example, the `CustomMercure` product model includes only GUI1, and GUI2 . The Factory Method that corresponds to the GUI variation point is `new_gui()`, so we add two stereotypes `<<GUI1>>` and `<<GUI2>>` to this factory method (see Figure 8).

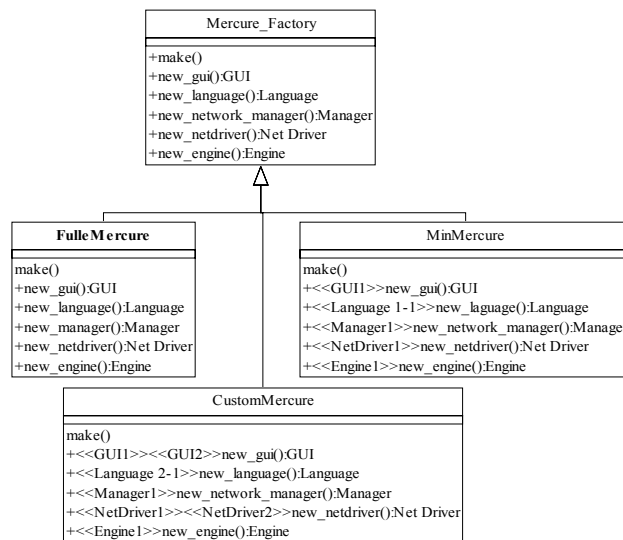


Figure 8. The Abstract Factory as a model decision for the Mercure PL

6.2. Product model derivation

Now we have to tackle with the automation of the derivation process exploiting the abstraction variability pattern and the decision model. The description of the derivation algorithm used to derive product models is illustrated in the Figure 9. It takes as input the PL model, and the Concrete Factory from the decision model and it generate as output the Product Model. It is decomposed in three steps: optional classes selection, model specialization, and the model optimization.

1. *Optional classes selection*: The first step consists to select optional classes using the concrete factory. For each factory method, we retrieve its stereotypes. These stereotypes define the names of the selected subclasses of the abstract class returned by the factory method. When the factory method does not define stereotypes (such as in the `FullMercure` concrete factory methods), all the optional sub classes of its return type are selected,
2. *Model specialization*: it removes all optional classes from the model, which have not been selected in 1. However, optional ancestors of selected optional elements are not removed,
3. *Model optimization*: it deletes unused factories and optimizes the inheritance (i.e when there is only one concrete class inheriting from an abstract one, the abstract class is omitted and replaced by the concrete one).

The product line model should satisfy generic constraints before the derivation and the product model derived should satisfy specific constraints. The generic constraints represent the pre-conditions of the derivation algorithm and the specific constraints represent the post-conditions:

```

DeriveProductLine(PL_model:Model, aConcreteFactory:Class)
  pre : -- check Generic Constraints on PL_model
  post :-- check Specific Constraints on the product model obtained
  
```

The Figure 10 illustrates the `CustomMercure` product model that we have obtained after derivation of the Mercure PL.

```

DeriveProductLine

Input: PL_model: Model
       aConcreteFactory: Class
Output : Product_model: Model

--Optional elements selection

Initiate selectedVariantsList to empty;
for each factory method in
  aConcreteFactory do
    initiate definedVariantsList to
      significant stereotypes of the factory;
    if definedVariantsList is empty
      then selectedVariantsList.add(all sub
classes of the returned type of the factory);
    else
selectedVariantsList.add(definedVariantsList) ;
    endif
  done

-- Model specialization

for each optional class C in PL_model do
  if (the class name of C not in
selectedVariantsList) and ( names of all sub
classes of C not in selectedVariantsList)
  then
    delete the class C from the PL_model;
  endif
done

-- Model optimization

delete all other factories;
optimize inheritance;
Product_model := PL_model;

```

Figure 9. Deriving a product line UML model

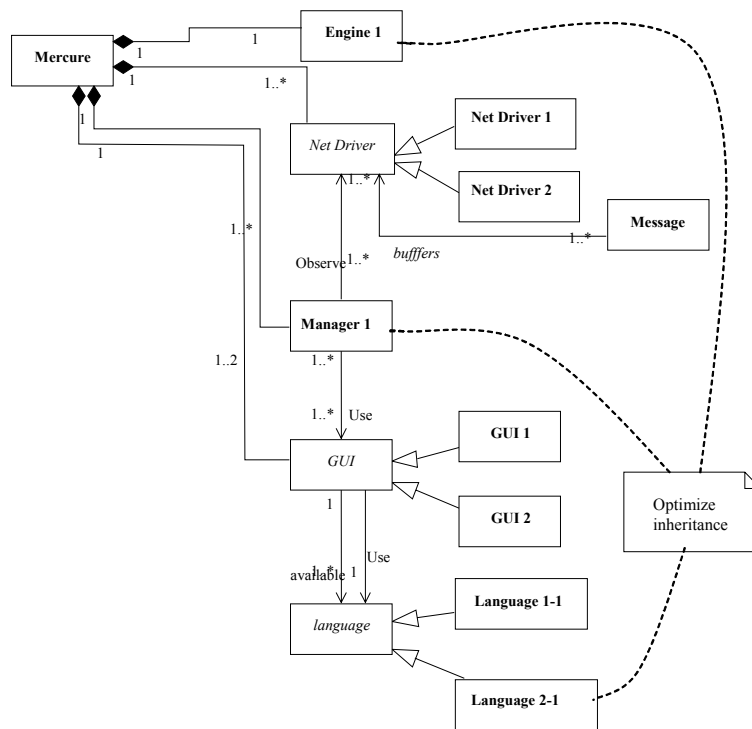


Figure 10. The CustomMercure Product UML model

7. Variability in the UML behavior models

In addition to the class diagram, UML includes other diagrams that describe other aspects of systems. The UML sequences diagrams can be used to model the behavior of the system. They are generally used to capture the requirements, but can then be used to document a system, or to produce tests. The UML 2.0 [23] makes sequences diagrams very similar to the ITU standard MSC (Message Sequence Chart) [7]. It introduces new mechanisms, especially interaction operators such as alternative, sequence, and loop to design respectively a choice of behaviors, a sequencing of behaviors and a repetition of behaviors. In this section, we use the example of the camera to illustrate how variability can be modeled in the UML 2.0 sequence diagrams. As in class diagrams, we show how variability can occur in sequence diagrams of a single product as well in the PL context.

1. *The Single Product Variability.* The UML 2.0 sequence diagram in Figure 11. called “Capture”, illustrates the interaction between the user and the cameras system to capture and store a data into the memory. The alternative (*alt*) operator is used to specify a choice between two behaviors. The first one uses the compressor to compress data before its store while the second one stores date without compression. So the alternative operator can be considered as a mechanism to specify variability; however in the same way as cardinality and class template mechanisms in the class diagrams, this variability concerns the single product an it is resolved after the product is delivered to customers.
2. *The Product Line Variability.* In the context of the camera PL, it is interesting to generate a set of requirements for each product. So we need to explicitly specify the variability in sequence diagrams; this variability is resolved when requirement for each product are derived and created. In [22], a first proposition to extend the MSC to support the specification of variability: three mechanisms are proposed: *virtual part*, *variation*, and *optionality*. Figure.12. shows the sequence diagram called “Capture_PL” that illustrates the capture requirement for the camera PL. The instance

Compressor is defined as optional using the stereotype <<optional>> (with a dashed life line). The *variation* mechanism has the common meaning in PL approaches: for a given product, only one alternative defined by the variation point will be present in the sequence diagram. The variation mechanism is depicted by means of rectangular frame, labeled by a variation name. Variants behaviors are separated by a dashed line. The example of Figure 12. contains a variation point Com, and two possible behaviors describing how data is stored depending on the presence of a compression device. The PL variability introduced in sequences diagrams allows us to generate requirements for all product line members. For example, the Capture sequence diagram for the product with no support for the compression is obtained by removing the instance Compressor (and all its incoming and outgoing messages are removed too) and the choice of the second (2) variant from the Capture_PL sequence diagram in Figure 12.

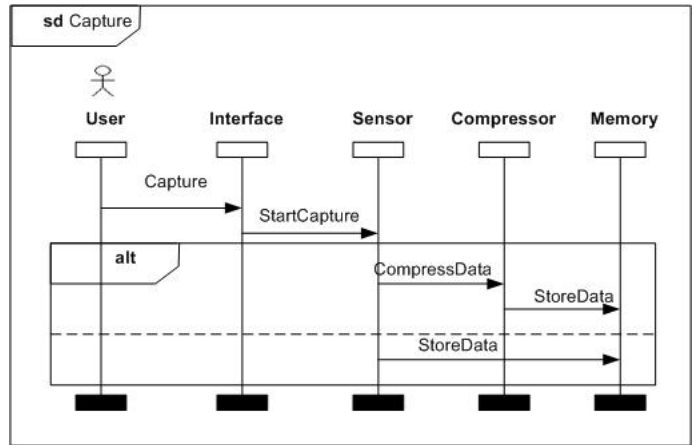


Figure 11. The “Capture” sequence diagram for the camera system (as a single product)

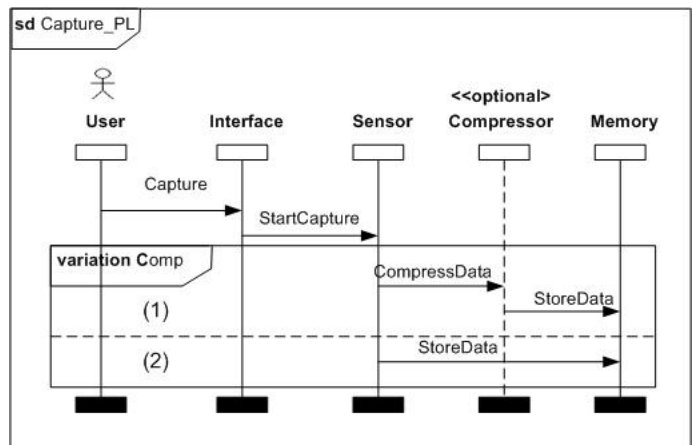


Figure 12 The “Capture_PL” sequence diagram for the camera PL

8. Related Work

As previously states, existing design, implementation techniques, and some approaches can be used for the variability management. As presented in [16,18], we briefly list some of these techniques:

- *Compilation techniques*: it is used to derive products at the compilation time by the inclusion or the exclusion of code segments during program compilation. For example, the conditional compilation can be used to manage variability at the compilation time.
- *Programming languages properties*: Object Oriented Languages offer some techniques such as inheritance, overloading, and dynamic binding that can be used to implement variability. Variation points are defined as abstract properties in the Product Line and each product defines these points in a specific way. Variability can also be implemented using class templates if the variants differ by a set of parameters.
- *Design patterns*: Design Patterns [3] can be used to model variability in software product line architectures. Patterns provide reusable solutions to certain types of problems and support the reuse of underlying implementations. In [12], the Abstract Factory pattern is proposed for reifying variants (this solution is detailed in the section 6). [1] proposes a set of patterns to model variability in product line architectures based on the notion of “Discriminants”.
- *Programming approaches*: some recent approaches of Software Engineering can be used for the variability management. Aspect-Oriented paradigm [5] is an engineering principle that aims at reducing systems complexity: it decomposes problems into a set of functional components and a set of aspects that crosscut functional components. Then it composes these components and aspects to obtain a system implementation. Some work [8,16,19] say that this approach can be used to implement variability. Aspects can be viewed as variation points, and product line members are specified by the aspects they contain. Generative Programming [13] is a software engineering paradigm based on the notion of “generator” for system families. Viability in Product Line can be managed by implementing components and generators as generic artifacts. A specific instantiation can be used to generate the implementation of a product.

The techniques presented above are generally related to programming languages. We also find some work [2,4,17] about the modeling of variability in the UML. These work mainly are based on the UML extensions mechanisms such as stereotypes and tagged values. Kobra [2] is a method that combines product line engineering and component-based software development. KobaA uses the UML to specify component. Variability is introduced to the Kobra component using stereotypes. The <<variant>> stereotype is added to all UML elements that represent variabilities in the family members. KobaA also introduces the decision model to document (in the form of a table) all variation points and its possible resolution.

9. Conclusion

We have proposed an approach based on the UML to model and to derive Product Line models. To achieve this, we propose the use of the UMLAUT framework [24] combined to the Model transformation Language (MTL).

UMLAUT is a framework for building tools dedicated to the manipulation of models described using the UML. A specific use is to apply a model transformation to an UML model, automating the derivation process then consists in writing the relevant model transformation. This transformation retrieves the useful model elements thanks to the selected concrete factory and then builds a specialized UML model corresponding to the selected Product. The challenge of such model manipulation is to be able to transform the model accessing its meta-level and ensuring the integrity of the derived model accordingly to the introduced specific constraints. A new version of the UMLAUT framework is currently under construction in the Triskell² team based on the MTL language, which is an extension of OCL with the MOF (Meta-Object Facility) architecture and side effect features, so it permits us to describe the process at the meta-level and to check OCL constraints (the generic constraints at first sight and specific constraints once the product model is derived). We present in appendix a detailed description of the derivation process as example of the MTL procedure.

The abstract factory derivation approach was described here for a specific PL, which is the Mercure project. It is possible to generalize this solution for others product lines that use the same abstraction variability pattern.

References

1. B. Keepence, M. Mannion, *Using Patterns to Model Variability in Product Families*, IEEE Software, 16(4): pages 102-108, 1999.
2. C. Atkinson et al., *Component-based Product Line Engineering with UML*, Addison-Wesley, 2002

² <http://www.irisa.fr/triskell/>

3. E., Gamma, R. Helm, R., Johnson, and J. Vlissides.. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995
4. ESAPS project deliverables. <http://www.esi.es/esaps/>
5. G. Kiczales, et al, *Aspect-Oriented Programming*, In ECOOP'97 –Object Oriented Programming 11th European Conference, 1997.
6. G. Sunyé, A. Le-Guennec, and J.M. Jézéquel, *Precise modeling of design patterns*, In LNCS, editor, Proceedings of UML 2000, volume 1939 of LNCS, pages 482--496, 2000.
7. ITU-T, *Z.120 : Message Sequence Chart (MSC)*, 1999.
8. J. Bayer, *Toward engineering product line using concerns*, GCSE 2000, Young Workshop, 2000.
9. J. Warmer, and A. Kleppe., *The Object Constraint Language – Precise Modeling with UML*, Object Technology Series. Addison-Wesley, 1998.
10. J-M. Jézéquel, *Object Oriented Software Engineering with Eiffé*.. Addison-Wesley. ISBN 1-201-63381-7, 1996
11. J-M. Jézéquel, *Object-oriented design of real-time telecom systems*, In IEEE International Symposium on Object-oriented Real-time distributed Computing, ISORC'98, Kyoto, Japan (April 1998).
12. J-M. Jézéquel, *Reifying Variants in Configuration Management*, ACM Transaction on Software Engineering and Methodology, pages 526-538, 1998.
13. K. Czarnecki, U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-wesley, 2000.
14. K. Kang, et al, *Feature-Oriented Domain Analysis Feasibility Study*, SEI Technical Report CMU/SEI-90-TR-21, November 1990.
15. L. Bass. P., Clements, and R., Kazman, *Software Architecture in practices*, Addison-Wesley, 1998.
16. M. Anastapoulos, C. Gacek, *Implementing Product Line Variability*, Technical report IESE report N°: 089.00/E, Franhofer IESE publication, 2000.
17. M. Clauß, *Modeling variability with UML*, In GCSE 2001 Young researchers Workshop. 2001
18. M. Svahnberg, J. Bosch, *Issues Concerning Variability in Software Product lines*, in F. van der Linden, editor, Software Architecture for Product Families International Workshop IW-SAPF-3, LNCS 1951, pp. 146-157, Springer 2000.
19. M.L. Griss, *Implementing Product-line Features by Composing Component Aspects*, in Proceedings of the First Software Product Line Conference, P. Donohoe, pp. 271-288, 2000.
20. Northrop.L., *A Framework for Software Product Line Practice–Version 3.0.*, http://www.sei.cmu.edu/pLDP/framework.html#framework_toc, Software Engineering Institute (SEI), 2002.
21. OMG. *UML specification*. Version 1.4, 2001.
22. T. Ziadi, L. Hélouët, J-M. Jézéquel, *Modeling Behaviors in Product Lines*, International Workshop in Engineering Requirement for Product Line (REPL'02), Essen, 2002.
23. U2 PARTNERS., *Proposal for UML 2.0 Infrastructure and Superstructure*, ad/00-09-01 and ad/00-09-02. <http://www.u2-partners.org/artifacts.htm>
24. W.-M. Ho, J-M. Jézéquel, A. Le Guennec, and F. Pennaneac'h, *UMLAUT: an extensible UML transformation framework*, In Proc. Automated Software Engineering, ASE'99, Florida, October 1999.

Appendix

A.1: OCL Auxiliary operations

```
context
ModelElement::isStreotyped(S : String): Boolean
post : result =
  self.stereotype →exists(s |
    s.name = S)

context
Namespace::presenceClass(C :String): Boolean
post : result =
  (self.oclIsKindOf(Class) and self.name = C)
  or
  (self.presenceClass(C))

context Class::AllSubClasses() : Set(Class)
post : result =
  self.specialization.child → iterate(c:Class; acc: Set(Class) = Set{} | acc →
including(c)→union(c.AllSubClasses()))

context Namespace::AllClasses() : Set(Class)
post : result =
  self.ownedElement → select(me: ModelElement |
  me.oclIsKindOf(Class)) → union (self.ownedElement. AllClasses())
```

A.2: A detailed description of the derivation algorithm

```
--Based on OCL extended with side effect features

ProductLineDerivation(aConcreteFactory:Class, pl:Model)
BEGIN

--Variant selection

Set(String) definedVariants
Set(String) selectedVariants
for op in
aConcreteFactory.feature→select( f: Feature
  | f.oclIsKindOf(Operation) and f.name.startsWith('new_') )
do
  Class opsReturnType :=
    ( op.parameter→select( p:Parameter | p.kind =
      #return) ).type
  definedVariants:= op.stereotype.name →
    intersection(
      opsReturnType.AllSubClasses().name)
  if definedVariants →isEmpty()
  then selectedVariants :=selectedVariant →
    union(opsReturnType.AllSubClasses().name)
  else selectedVariants :=selectedVariant →
    union(op.stereotype.name)
  endif
done

--Model specialization

for C:Class in pl.AllClasses()
do
  if (C.isStreotyped('optional')) and
    (selectedVariant→excludes(C.name)) and
    selectedVariant→
      excludesAll(C.AllSubClasses().name)
  then
    deleteElement(C, pl)
  endif
done
```

-- Model optimization

```
aConcreteFactory.generalization.parent.specialization.child→  
excluding(aConcreteFactory)→collect(cf : Class|  
  deleteElement(cf, pl))
```

optimizeInheritance(pl)

END

A.3: The Dependency and the generalization meta-classes in the UML meta-model

