Product Line Derivation with UML [1]

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

Handling the various derivations of a product can be a daunting (and costly) task. To tackle this problem, we propose a method based on the use of a creational design pattern to uncouple the variations from the selection process. This makes it possible to automatically derive a given product from the set of all possible ones, and to specialize its model accordingly. The contribution of this paper is to provide a set of patterns for modeling variability issues of a Product Line Architecture to define architectural constraints for Product Line expressed in UML as meta-level OCL constraints and to propose an approach to automate the derivation process.

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) [19], 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 derivation activity and 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 the Mercure PL. In section 3, we propose some mechanisms to specify variability in the UML class diagrams. Section 4 presents PL constraints and their specification with the OCL, and the section 5 illustrates the derivation process. Finally section 6 concludes this work.

2. Background in Product Line Engineering

2.1. The Software Product Line approach

The general process of Product Line Engineering, as found in the literature [4,5,18], 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 [4].

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) [13] 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

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

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.

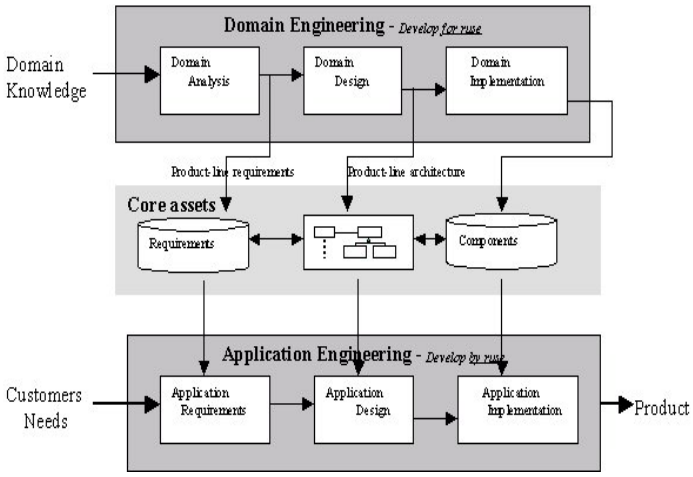

Figure 1. The general process for Product Lines Engineering

2.2. The Software Variability Management

The main challenge in the context of software product lines is to model and implement the variability. Even if the product line approach is a new paradigm, managing variability in software systems is not a new problem, and it can be solved by some existing approaches. [14,16] study how existing techniques can be used for the variability management. 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 [8] 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 (we will present in more detail this solution in section 5). [2] 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 [6] 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 [9,14,17] 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 [4] 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 [3,5,15] about the modeling of variability in the UML. These work mainly are based on the UML extensions mechanisms such as stereotypes and tagged values. We will present in the next section mechanisms that we have used to specify variability in UML class diagrams.

2.3. The Variability in the Mercure PL

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.

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). To identify variabilities in the Mercure PL, we specify its domain model using FODA notations, slightly modified and extended by [4]. 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.

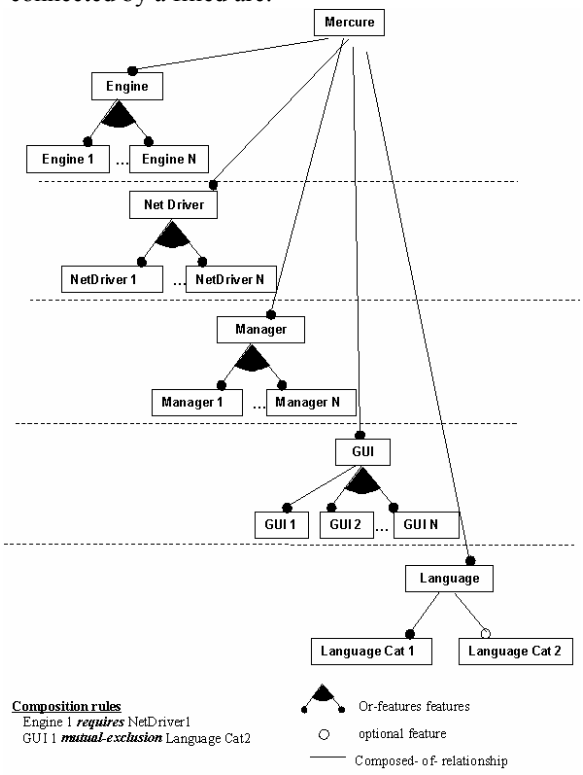

Figure 2. The FODA diagram for the Mercure PL

Figure 2. 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 first one and the second one is 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 Cat 2 (see figure 2.)

3. Variability in UML class diagrams

The Unified Modeling Language (UML) [19] is a standard language for the object-oriented analysis and design. It defines a set of notations to describe different aspects of systems. In this section, we present three mechanisms that can be used to specify the variability in the UML class diagram: *Abstraction*, *Parameterization*, and *Optionality*.
Abstraction: Using an object-oriented analysis and design approach, it is natural 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).
Parameterization: the UML classes can be defined as generic assets with a set of parameters; each product binds these parameters in a specific way. UML class templates can be used as parameterization classes.
Optionality: the Product Line model includes all elements associated to all products, so in specific products some of these elements called "optional" can be omitted. To show optionality, we use an ad-hoc stereotype «optional» that can be applied to classes, packages, and interfaces.

The UML class diagram in the figure 3 represents the Mercure PL model. It basically says that a Mercure system is an instance of the MERCURE class, aggregating an ENGINE (that encapsulate 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.
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 (Abstraction variability mechanism) 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.

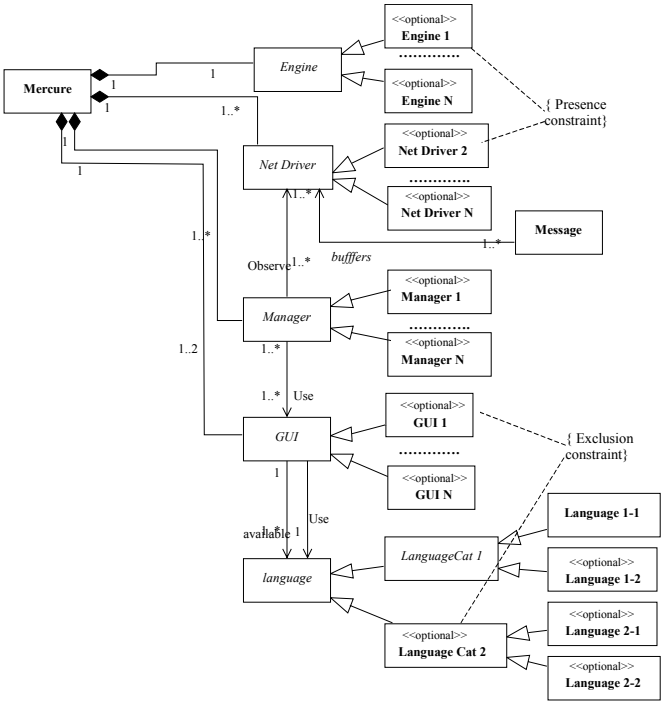

Figure 3. The Mercure Product Line UML class diagram

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

4. Managing the PL constraints

[1] 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 [15] use UML stereotypes to show dependencies between classes.

The Object Constraints Language (OCL) [23] 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.

4.1 The Generic Constraints

The introduction of variation points, 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 [7] 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 4.

The generic constraints may be seen as well-formedness rules for the UML modeled product lines.

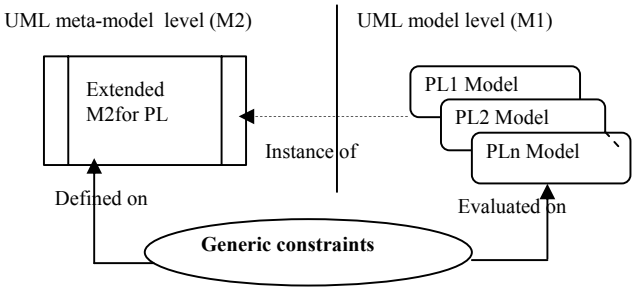

Figure 4. Generic constraints as OCL meta-level constraints

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 specify a require relationship between two or more elements. It is represented in the UML meta-model [19 p 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’s 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 [19 p 2.15], 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* [19 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.

4.2 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 5.

The specific constraints are parts of the PL model definition.

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 [19 p 2.189]. A Model is a namespace that contains a set of *ModelElement* whose names designate a unique element within the namespace.

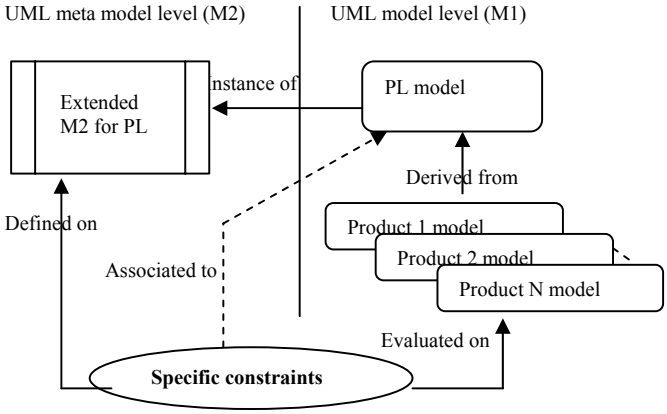

Figure 5. Specific constraints for PL model as OCL meta-level constraints

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 NETDRIVER2 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('NETDRIVER2')
```

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 LANGUGE_CAT2**,** 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
LANGUGE_CAT2 cannot be present in the same model
inv:
(self.presenceClass('GUI1') implies not
self.presenceClass('LANGUGE_CAT2'))and
(self.presenceClass('LANGUGE_CAT2') implies not
self.presenceClass('GUI1'))
```

In the UML class diagram (see figure 3.), we use graphical shorthands to show the above constraints.

5. 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 [5]. 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.

5.1. The decision model in a Product Line

In [12], the creational design pattern *abstract factory* [8] is used to refine the several variation points. This process is easily customizable by defining an interface for creating variants of Mercure's five variation points (Engines, Net Drivers, Managers, GUIs and Languages). Obtaining an actual variant of the Mercure PL then consists in implementing the relevant concrete factory. The idea is originally used to simplify the Software Configuration Management by reifying the variant of an object-oriented software system into language-level objects. Our aim in this section is to use this idea as a design of the PL decision model.

The decision model of the Mercure PL is illustrated in the figure 6. Each concrete factory is related to one product in the Mercure PL, and each creational operation in the different concrete factories corresponds to a variation point. We use stereotypes to restrict the returned type of creational operations to the possible one. For example, the product model corresponding to the concrete factory CustomMercure includes only GUI1, and GUI2 classes as GUI variants. So we add two stereotypes <<GUI1>> and <<GUI2>> to the operation new_gui().

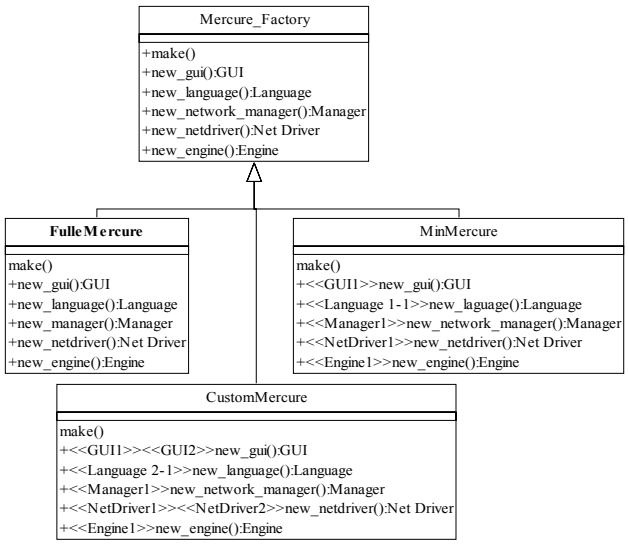

Figure 6. The Abstract Factory as a model decision for the Mercure PL

5.2. Product model derivation

At this stage, we have precisely defined the Product Line, 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 transformation algorithm used to derive product models is illustrated in the figure 7. The transformation algorithm is decomposed in three steps: variants selection, model specialization, and the model optimization.

1. The variants selection: Variation points are defined by return types of concrete factory operations. The selected variants are defined by their significant stereotypes (as names of variants). When the operation does not define stereotypes (such as in the FullMercure concrete factory operations), all sub classes of its return type is selected,
2. the model specialization: it removes all optional classes from the model, which have not been selected in 1. However, optional ancestors of selected variants are not removed,

3. the model optimization: it deletes unused factories and optimize the model (i.e when there is only one concrete class inheriting from an abstract 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 transformation operation and the specific constraints represent the post - conditions:

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

The figure 8 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

--Variants selection

 Initiate selectedVariantsList to empty;
for each factory operation in
  aConcreteFactory do
  initiate definedVariantsList to
    significant stereotypes of the operation;
  if definiedVariantsList is empty
    then selectedVariantsList.add(all sub
classes of the returned type of the operation);
    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 7. Deriving a product line UML model

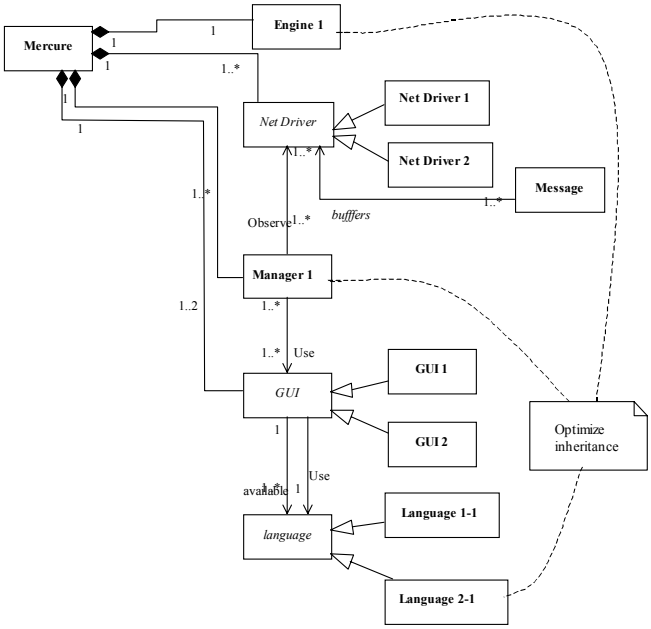

Figure 8. The CustomMercure Product UML model

6. Conclusion

We have proposed an approach based on the UML to model and to derive Product Line models. This approach especially focuses on static models represented by the UML class diagrams. To achieve this, we propose the use of the UMLAUT framework [22] 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 Triskell[2] 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

[2] http://www.irisa.fr/triskell/

constrains 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.
As future work, we want to implement a UML profile for Product Line (including behavior aspects as proposed in [21]). This UML profile defines a set of stereotypes and a set of generic constraints to ensure any PL correctness. The user PL specification includes a set of models enriched by specific constraints, which may guide the derivation process. The derivation consists in applying a transformation algorithm written in MTL.
The abstract factory derivation approach was described here for a specific PL, which is the Mercure project. We think that it's possible to generalize this solution for others product lines that use the same abstraction variability pattern.

7. References

1. Bass, L., Clements, P., and Kazman, R. *Software Architecture in practices*, Addison-Wesley, 1998.
2. B. Keepence, M. Mannion, "Using Patterns to Model Variability in Product Families", IEEE Software, 16(4): pages 102-108, 1999.
3. C. Atkinson, J. Bayer, and D. Muthig, "Component-based product line lopment. the KobrA approach", In Proc. of the 1st Software Product Lines Conference (SPLC1), pages 289–309,2000.
4. Czarnecki K., Eisenecker U.W., *Generative Programming: Methods, Tools, and Applications*, Addison-wesley, 2000.
5. ESAPS project deliverables. http://www.esi.es/esaps/
6. G. Kiczales, et al, "Aspect-Oriented Programming", In ECOOP'97 –Object Oriented Programming 11th European Conference, 1997.
7. 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.
8. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995
9. J. Bayer, "Toward engineering product line using concerns", GCSE 2000, Young Workshop, 2000.
10. Jézéquel, J.-M.. *Object Oriented Software Engineering with Eiffe,*. Addison-Wesley. ISBN 1-201-63381-7, 1996
11. J-M. Jézéquel, "Object-orented 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. Kang.k. et al Feature-Oriented Domain Analysis Feasibility Study, SEI Technical Report CMU/SEI-90-TR-21, November 1990.
14. M. Anastapoulos, C. Gacek,, "Implementing Product Line Variability", Technical report IESE report N°: 089.00/E, Franhofer IESE publication, 2000.
15. M. Clauß, "Modeling variability with UML", In GCSE 2001 Young researchers Workshop. 2001
16. 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.
17. 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.
18. 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.
19. OMG. UML specification. Version 1.4, 2001.
20. Pierre America and Steffen Thiel and Stefan and Martin Mergel, Introduction to Domain Analysis, ESAPS project, 2001 web = http://www.esi.es/esaps/.
21. T. Ziadi, L. Hélouët, J-M. Jézéquel, "Modeling Behaviors in Product Lines", International Workshop in Engineering Requirement for Product Line (REPL'02), Essen, 2002.
22. 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.
23. Warmer, J., and Kleppe, A.. *The Object Constraint Language – Precise Modeling with UML*, Object Technology Series. Addison-Wesley, 1998

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.isStereotyped('optional')) and
  (selectedVariant→exludes(C.name)) and
  selectedVariant→
       exludesAll(C.AllSubClasses().name)
 then
   deleteElement(C, pl)
 endif
done

-- Model optimization

aConcreteFactory.generalization.parent.specializ
ation.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