



**Fraunhofer** Institut  
Experimentelles  
Software Engineering

# System Family Architecture Description Using the UML

Final Report

**Author:**  
Oliver Flege

This work has been partially funded  
by the ESAPS project (Eureka  $\Sigma!$  2023  
Programme, ITEA project 99005)

IESE-Report No. 092.00/E  
Version 1.0  
December 15, 2000

---

A publication by Fraunhofer IESE



Fraunhofer IESE is an institute of the  
Fraunhofer Gesellschaft.

The institute transfers innovative software  
development techniques, methods and  
tools into industrial practice, assists com-  
panies in building software competencies  
customized to their needs, and helps them  
to establish a competitive market position.

Fraunhofer IESE is directed by  
Prof. Dr. Dieter Rombach  
Sauerwiesen 6  
D-67661 Kaiserslautern



## Executive Summary

At IESE, we developed a method for system family architecture description using the UML that is characterized by its focus on simplicity, automation possibilities, and completeness. First, the representation and other modeling concepts are fully UML compliant. This supports simplicity and allows for the usage of existing UML modeling tools. Second, we investigated not only representation, but also evaluation and instantiation aspects. Since these aspects are closely related and interdependent, we think that omitting only one of them would severely reduce the usefulness of the overall approach. Third, our solution relies on being part of a product line infrastructure, and in particular on the possibility of interfacing with a decision model. By doing so, we were able to further reduce the complexity of our approach and to focus on the issues that are unique to modeling family architectures in the UML.



## Table of Contents

<b>1</b>	<b>Context and Objectives</b>	1
1.1	Architectural Variability Mechanisms	2
1.2	Guidelines for Using Variability Mechanisms	3
1.3	Decision Models	4
<b>2</b>	<b>Solution</b>	5
2.1	Rationale	5
2.2	UML Assets	7
2.2.1	Static Diagrams	7
2.2.2	Dynamic Diagrams	7
2.2.3	Relation to Decision Model	8
2.3	Usage of the Architecture	10
2.3.1	From Domain Design to Application Design	10
<b>3</b>	<b>References</b>	13



# 1 Context and Objectives

In our work with industrial partners, we observed that more and more companies create models of their software systems. The predominant notation — at least in the domain of information systems — is the Unified Modeling Language (UML). The need for modeling is related to the increasingly complex structure and behavior of software systems. In addition to that, it is acknowledged that architectural models provide useful abstractions that facilitate the understandability and maintainability of a system. The widespread use of modeling is supported by the availability of sophisticated UML-modeling tools. Besides pure modeling, most of these tools also support “round-trip” engineering (i.e., using the model to generate (parts of) the code and, vice versa, parsing code to create/update a model), which helps narrowing the gap between the model and the code and thus to further raise acceptance and usage of modeling.

However, companies that model their systems with UML face the problem that as soon as they start thinking in terms of product lines, they find that neither the UML nor any commercial modeling tool provides ready-to-use means for dealing with variabilities. Although some simple workarounds may be feasible for modeling (e.g., using presentation styles or stereotypes to indicate variability), these “solutions” do not scale well, do not support instantiation, and are not at all integrated with round-trip engineering capabilities. The efficient (i.e., automatic) instantiation of a product line architecture in order to derive an instance-specific architecture is currently not supported by mainstream modeling notations, methods, and tools. Nevertheless, it is not sufficient to be able to describe commonalities and variabilities in a generic architecture model. Without means for automatic instantiation, such a model becomes more and more difficult to understand and maintain the more variability it contains. Furthermore, since instantiation changes the architecture, it is by no means always obvious whether all possible instances would constitute valid architectures. It is therefore necessary to define a set of evaluation rules that can be checked automatically to ensure this property.

The objectives of our work are to help solving the problems mentioned above by providing methods that support

- modeling of product line architectures using the Unified Modeling Language,
- evaluating those product line architectures with respect to variability information,
- instantiating those product line architectures

In order to reach those objectives, we had to address and solve the following problems:

- How can variabilities be incorporated and represented in a UML model that is compliant with the UML standard (i.e., contains no custom elements and only makes use of the UML's built-in extension mechanisms)? Compliance with the standard is of utmost importance to be able to use existing tools and to avoid acceptance problems.

Addressing this problem brings about the following sub-problem: Which kinds of variabilities do we have to represent in a generic UML model and how can we best represent each of them?

- What are properties of a UML product line architecture model that can be used to determine whether the model is well-formed with respect to variability information (i.e., how can we make sure that every possible instantiation of such a model yields a well-formed UML model)?
- How can the variability information stored in the generic architecture model be used to support automatic instantiation?
- According to our understanding, the decisions that are related to variabilities in the product line architecture should be encapsulated in a dedicated asset, the decision model. Another problem, which is orthogonal to the ones mentioned above, is therefore: How can a generic architecture model be connected to a decision model and which services does the decision model have to provide to support modeling, evaluation, and instantiation?

These problems cannot be solved in isolation, because there are some straightforward relations: The kind of information stored in the decision model is the basis for deciding how this information can be referenced in the UML model. On the other hand, the representation, instantiation, and evaluation needs bring about requirements on the kind of information that should be stored in the model.

We see our work in the ESAPS project as a first step towards the tool-supported description of product line architectures (PLAs). Therefore, our solution is oriented towards easy integration with existing tools (i.e., UML modeling tools) as well as with tools that we think are a necessity in a product line development environment (i.e., a decision model implementation).

## 1.1 Architectural Variability Mechanisms

The first problem to be solved is to determine which kinds of variabilities we have to represent in a generic UML model. Our original idea was to collect a set of the most relevant architectural variability patterns. However, an analysis of several industrial projects has shown that these patterns do not seem to exist. First of all, the variabilities we encountered were spread over all possible levels of detail. A result being fairly in line with Basset's observation that "Effective software reuse involves components of all size scales, down to bits" [1]. Given

this situation, it is very difficult — if not impossible — to distinguish between *architectural* variability mechanisms and “others”, the latter ones being identified by appearing only on lower levels of abstraction. First and foremost, this problem is hard to solve because there are different opinions on how “deep” an architecture should be documented. But even if it were possible to elicit architectural variability patterns, it would still be desirable to develop a unified and consistent mechanism for dealing with variabilities that can be used at all levels of abstraction. Therefore, we decided to look at variability mechanisms in general.

Although distinguishing variability mechanisms by the level of abstraction on which they are used is not practical, it is quite important to distinguish variabilities based on their binding time, which can be either construction-time or run-time. The former includes every time different than run-time (e.g., compile-time, link-time). Construction-time variabilities depend on decisions that are resolved before the system is finally constructed and are immutable at run-time. In contrast, run-time variabilities are the ones that are incorporated in the final product. Only construction-time variabilities result in different products and are thus essential for developing product lines. Therefore, our focus is exclusively on construction-time variability. A reference architecture describes the pre-construction-time view, whereas a derived instance architecture describes the post-construction-time view. Note that the emphasis on construction-time variability is consistent with Basset's premise that “Effective reuse involves construction-time variability that is sensitive to differing context of use” [1].

Another important restriction is that we focus exclusively on variabilities with a complete set of specified variants. In other words, we discard variabilities that merely consist of a variation point that might be used by some application developer in some unanticipated way. The reason for this restriction is that unspecified variability has no impact during instantiation and evaluation of a reference architecture and is therefore not considered to be a problem in the realm of reference architecture modeling.

## 1.2 Guidelines for Using Variability Mechanisms

We are not providing any guidelines, because there are just too many factors that influence the “right” or “best” choice of a variability mechanism such as the domain or scope of the product line, the kind of project, or the capabilities provided by the development environment or the programming language.

### 1.3 Decision Models

In order to understand the concept of a decision model, it is necessary to define a number of terms. A product line architecture, as well as any other generic asset<sup>1</sup> in a product line infrastructure, refers to *variabilities* among product line members. An example for variability is "Some products in the PL are multithreaded, others are not". Each variability embraces a number of *variants* (e.g., variant<sub>1</sub>: "the product is multithreaded", variant<sub>2</sub>: "the product is not multithreaded"). Thus, each variant can be seen as an instance of a certain variability and each variable element in a generic asset actually belongs to a certain variant (e.g., an architectural component for scheduling threads would belong to variant<sub>1</sub>).

Each application in the product line can be characterized by the specific variants it includes. This is commonly done by resolving a set of open *decisions* (or choices). Each of those decisions is related to a certain variability (e.g., decision D<sub>1</sub>: "Is the application multithreaded?") and has an associated set of possible *resolutions*, which in turn are related to certain variants (e.g., resolution D<sub>1,1</sub>: "yes" → variant<sub>1</sub>, resolution D<sub>1,2</sub>: "no" → variant<sub>2</sub>). Usually, most of the decisions have an impact on several generic assets. It is therefore advisable to maintain all decisions in a separate decision model instead of modeling them separately for and in each generic asset.

A decision model contains a collection of open decisions, defines the possible resolutions for each open decision, includes rules that specify relations between decisions (e.g., "if decision A is resolved to a<sub>1</sub>, then decision B can only be resolved to b<sub>1</sub> or b<sub>3</sub>"), and usually provides some means for structuring decisions to increase the usability. In a *decision model instance*, each decision has actually been made, that is, one of the predefined resolutions for each decision has been chosen.

1 A *generic asset* is any product that describes a certain aspect of some or all product line members in terms of their commonalities and variabilities.

## 2 Solution

### 2.1 Rationale

The proposed modeling of variable elements in a reference architecture description does not require an extension or other modification of the UML metamodel, because hardly any modeling tool would be able to support these kinds of changes. Instead, we tried to investigate the simplest solution possible, which relies only on standard UML capabilities.

The representation of optional elements is based on a dedicated presentation style, which includes specific line, fill, and font colors. In addition to that, each optional element is given the specific stereotype «Optional» in order to facilitate easy identification of optional elements in black and white printing. However, using a stereotype for this purpose can be problematic, because elements of the same type but with different stereotypes can have different graphical representations in the UML. That is, changing the stereotype might cause a change of the entire graphical representation of an element instead of just adding a word in guillemets (“«»”). Therefore, we only apply the «Optional» stereotype to elements that do not have any other stereotype assigned. An example of an optional class and its representation is shown in Figure 1 on Page 6.

In addition to marking model elements that are directly subject to variability, we also found it useful to denote that an otherwise non-optional element *contains* at least one optional element. For this purpose, we use the «HasVariability» stereotype together with another dedicated presentation style. In the example shown in Figure 2 on Page 8, the state “FooBar” contains optional sub-states. Again, this stereotype is only used if an element does not have any other stereotype yet. One problem associated with this feature is how it should be propagated in a containment hierarchy (e.g., should a package containing a package that contains an optional class be decorated with the «HasVariability» stereotype?). Since it would be very easy to restrict this kind of propagation to an arbitrary number of hierarchy levels in a tool by using a parameter, it is probably the best to leave this decision to the tool users.

Of course, purely representational means are not enough to describe variability in a reference architecture. It is not very helpful to know that some architectural element is optional without knowing to which variability it belongs and for which variant(s) it is supposed to be part of an instance architecture. Therefore, this additional information has to be stored somehow with or preferably *in* the UML model. The best way we found to achieve this is to make use of tagged

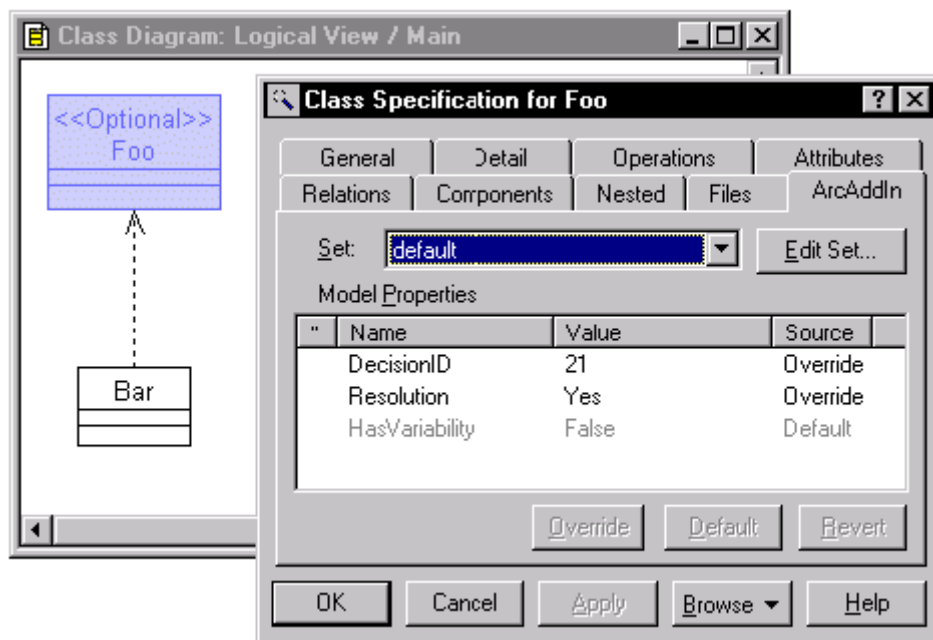
values, one of the UML's built-in extension mechanisms. Our solution includes a number of tagged values to store the following information:

- A reference to a decision in the decision model (by means of a unique ID).
- The required resolution of that decision upon which the optional element would be part of an instance architecture.
- A flag that indicates whether an element contains variable elements (e.g., whether a class contains optional attributes, methods, or nested classes, or a package contains optional classes). This information is cached in the tagged value in order to avoid costly computations, and is of course only used for those elements that are actually capable of containing other elements (i.e., classes, packages, subsystems, states, and activities).

An example of how a tool can maintain a set of tagged values is shown in Figure 1. Of course, a user of a PLA modeling tool cannot be expected to manipulate those tagged values manually. According to our envisioned cooperation between the PLA modeling tool and a decision model implementation, which is described in Section 2.2.3, the manipulation of those values is completely hidden by the tools and the user does not even have to be aware that the tagged values exist at all.

According to the UML specification "any modeling element may have arbitrary attached information in the form of a property-list consisting of tag-value pairs", just as any modeling element may have a stereotype. Therefore, our proposed solution is essentially applicable to any model element. It should be noted, however, that most commercial modeling tools only support tagged values for a subset of those elements.

Figure 1:  
Variability  
Representation and  
Realization



For sake of simplicity, the only type of variability currently supported by our solution is optionality, and each optional model element has to be connected to a decision and a specific resolution individually. The concept of an alternative is not explicitly supported. Instead, we assume that the decision model provides it by including a decision that allows for a set of alternative resolutions, which may or may not be mutually exclusive. One potential drawback of this limitation is that alternative model elements are not distinguishable at first sight in the reference architecture, that is, it is not obvious that optional model elements represent different variants of the same variability. For an example, see Figure 2 in which two transitions leave the state "Foo" — it is not obvious that these transitions belong to the same decision that has mutually exclusive resolutions and thus represent an alternative. A very simple first step towards a solution would be if a tool allowed the user to specify different representation options (i.e., line, fill, and font colors) for each decision.

## 2.2 UML Assets

Our proposed solution does not involve different kinds of variability concepts for different kinds of diagrams or model elements. In other words, each model element (i.e., an instance of the class *ModelElement* in the UML Metamodel) can be made optional in the same way by connecting it to a decision/resolution tuple in the decision model.

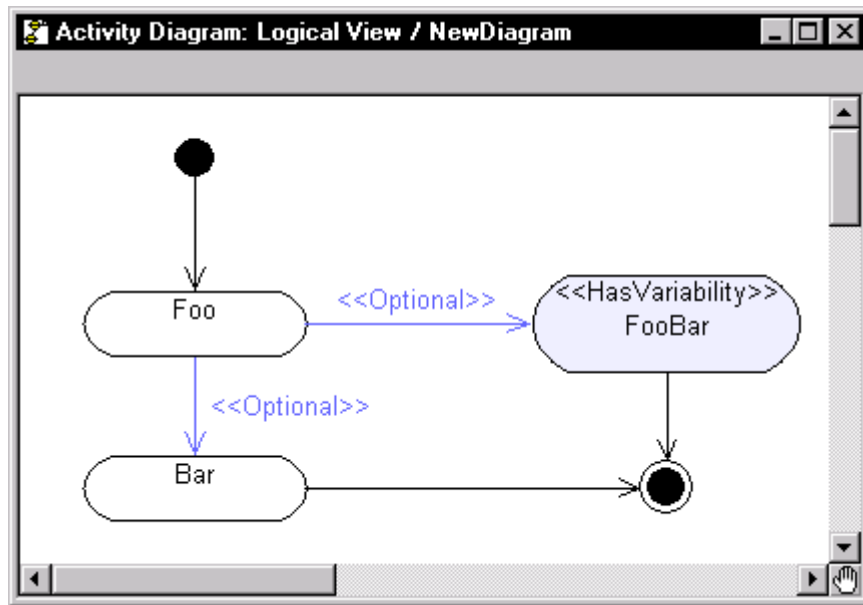
### 2.2.1 Static Diagrams

See Figure 1 for an example. The other static diagrams are treated similarly.

### 2.2.2 Dynamic Diagrams

See Figure 2 for an example. The other dynamic diagrams are treated similarly.

Figure 2:  
Statechart Diagram  
with Variability



### 2.2.3 Relation to Decision Model

The decision model is an essential element in our approach. Since we see our work as a first step towards tool supported PLA modeling, we are now going to describe how an implementation of a decision model could provide services to a PLA modeling tool.

In order to define how a decision model implementation could interact with other product line tools, it is helpful to look at the problem from an object-oriented perspective and thus in terms of responsibilities. The responsibility of a decision model is to control the decisions that govern variability in the product line and to facilitate consistent instantiation. It is, however, not responsible for actually applying the variability information in the different assets (e.g., instantiating them), because this would violate the principles of encapsulation and information hiding and would consequently make it impossible to define a decision model that could be combined with arbitrary generic assets.

A UML modeling tool that is supposed to support PLA development is responsible for

- representing variable model elements visually distinguishable from common ones,
- evaluating the consistency of the variability information (i.e., whether or not all possible instances would be well-formed UML models), and
- instantiating the reference architecture model.

This leads to three different categories of services that should be offered by a decision model implementation: support for modeling, support for evaluation, and support for instantiation.

**Support for Modeling.** Support for modeling is about connecting variants of variabilities in a PLA with a resolution of a decision in the decision model. A decision model should therefore offer the following services:

- Select a decision and a particular resolution of that decision. Upon selection, the decision model returns an identifier<sup>1</sup> for the decision and a string encoding the required resolution. This service is invoked from the PLA modeling tool every time the user selects a set of model elements and requests to make them optional. The data returned by the decision model is then stored in the appropriate tagged values of all selected model elements.
- Show information about a particular decision, such as its description, resolution possibilities, and relations to other decisions. This service is invoked whenever the user selects an optional element in the PLA model and requests information about the associated decision.

An example of how the information from the decision model can be used and stored in a UML model is shown in Figure 1. The class diagram shows two classes, one of which is optional (Foo). The tagged values, which are shown in the class specification for Foo, link the class to the decision model. In the example, the presence of Foo in a model instance depends on a decision model instance in which the decision with the ID 21 has been resolved to "Yes".

**Support for Evaluation.** Support for evaluation is about being able to answer the question whether all possible resolutions of the decision model would result in a valid instance of the PLA. For example, consider a UML class diagram containing a class **A** and its subclass **B**, both of which are optional. In any valid instance, presence of **B** implies presence of **A**, which means that the selection of the option connected to **B** implies the selection of the option connected to **A**. Another example can be found in Figure 1. It actually shows an invalid model, because class Bar depends on the optional class Foo, which might not be present in certain instances.

In general, the decision model has to offer services that allow to answer the following question: Given a decision model in which the decisions  $d_1, \dots, d_m$  are resolved as  $r_1, \dots, r_m$ : Is it possible to resolve decision  $d_n$  to  $r_n$  or to resolve  $d_n$  in any other way than  $r_n$ ? In order to answer this question automatically, the following services are needed:

<sup>1</sup> I use the term identifier to refer to any kind of information that enables a tool to select a specific object. An identifier may be a true object reference, but may also be a simple string or a numeric value. In general, simpler types promise easier integration of tools.

- Create a new decision model instance (DMI) without actually resolving any open decision. An identifier for the DMI is returned to the PLA modeling tool that will be used in subsequent calls to resolve some decisions within this specific DMI and thus to create the precondition for the question to be answered.
- Resolve a particular decision in DMI in a particular way (e.g., resolve the decision associated with class **B** to the value required by **B**, so that **B** will be part of the PLA instance to be tested). The decision model returns whether the resolution was successful or not.
- Check whether a particular decision in DMI can be resolved in a specific way (e.g., whether the option associated with **A** can be resolved to ensure **A**'s presence in the instance). The decision model returns yes or no.
- Check whether a particular decision in DMI can be resolved in any other than a specified way. The decision model returns yes or no.
- Delete DMI. This indicates that the specific DMI is no longer needed and can be discarded by the decision model.

**Support for Instantiation.** Support for instantiation is about deriving an instance of a PLA. The services required to support this are:

- Select a decision model instance DMI. Upon selection, the decision model returns an identifier for the specific instance to the PLA modeling tool. It might also be useful to be able to select a partially instantiated decision model (i.e., with some decisions still open).
- Check whether a given resolution is matched by the actual resolution of a particular decision in DMI. The decision model returns either yes, no, or that the decision has not yet been made in DMI. The given resolution and the decision originate from an optional element in the PLA with which they had been associated. For a complete instantiation, this service of the decision model has to be called for every optional element in the PLA.

## 2.3 Usage of the Architecture

### 2.3.1 From Domain Design to Application Design

**Instantiation.** The instantiation of a PLA (or any other generic asset) based on a decision model instance is straightforward, if we assume that each optional element in a PLA references a particular resolution of a decision in the decision model. Those optional elements (i.e., variants of a variability) will be included in the PLA instance whose associated resolution matches the actual resolution in the decision model instance.

As the first step in the instantiation process, a decision model instance has to be selected. This instance determines the actual resolutions of all decisions that are

referenced by optional model elements. Then, the following algorithm, which is easy to automate, can be used to instantiate the PLA.

- For all packages in the reference architecture:
  - If the package is optional, check whether its required resolution is matched by the actual resolution of its associated decision. If this is not the case, delete the package from the model.
  - Since a package “owns” its contained model elements, the deletion of the package also includes the deletion of those elements. Additionally, the deletion of the package deletes all the dependencies that exist between the package and other model elements.
- For all remaining model elements that are not packages:
  - If the element is optional, check whether its required resolution is matched by the actual resolution of its associated decision. If this is not the case, delete the element from the model. This deletion will also affect all relations between the element and other elements, as well as all elements contained in the element.

**Evaluation.** In our approach, instantiation is essentially concerned with removing those parts of the reference architecture that do not belong to a specific instance. We do not support any kind of generation that would lead to new elements added to the reference architecture. Thus, a fundamental question is, whether the validity of the architecture is maintained during instantiation. In other words, whether the deletion of elements from a reference architecture preserves the original concepts/intentions/etc. of the PLA. In this context, it is not sufficient to look at the instance architecture and determine whether it is well-formed as a whole. Additionally, we have to make sure that the properties of model elements that remain in the instance architecture remain unchanged during instantiation. In the following, we will give examples of both of these evaluation tasks.

**Local Evaluation.** Local evaluation is concerned with the effects that instantiation has on a single model element that has its own set of dependencies to other model elements. Whenever an instantiation could break at least one of those dependencies, this indicates a problem.

In general, local evaluation is concerned with the following situation: a model element **A** is connected to some other element **B** in such a way that **A** depends on **B**. Table 1 shows some of the dependencies that we investigated.

Table 1:  
UML Element  
Dependencies

Element A	Element B	Dependency
Class	Class	A is a specialization of B (Inheritance)
Class	Interface	A realizes B
Class	Class	A uses B
Class	Class	A depends on B
Class	Class	A is associated with B and navigation from A to B is possible
Package	Package	A depends on B («imports» or «access» dependency)
Package	Package	A is a specialization of B (Inheritance)
State	State	Transition from A to B
...	...	...

A problem with these dependencies arises as soon as the element **A** depends on is subject to variability. In that case, it might happen that **A** but not **B** is present in an instance, which would be an error. In order to detect this kind of problem, the following evaluation has to be performed: For each decision model instance that would lead to **A** being present in the resulting model instance, check if it could happen that **B** would not be present. A practical approach would be the following, that is based on the decision model services described in section Support for Evaluation:

- 1 Create an empty decision model instance (i.e., without resolving any decision)
- 2 For each optional model element that has to be part of the model to ensure **A**'s presence in the model (i.e., **A** itself and all packages that either include **A** directly or contain a package that contains **A**), resolve the related decisions to ensure that **A** is present. Unfortunately, there may be many such configurations depending on the resolutions required by the optional elements. Two examples for a required resolution that can be satisfied in a number of different configurations are: The set of chosen alternatives has at least two members; the set of chosen alternatives includes a particular subset of members. In such a situation, it would be necessary to perform the evaluation for all configurations that are possible.
- 3 Test whether it would be possible to resolve the decisions that are related to **B**'s presence in the instance architecture in such a way that **B** would not be present. In a well-formed model, this should not be possible (either, the decisions have already been resolved in different ways to ensure **A**'s presence, or they are constrained by those resolutions).
- 4 Again, there may be many possible ways to prevent **B** from being present in an instance architecture; to show that a reference architecture is not well-formed it is sufficient to find one troublesome configuration.

**Global Evaluation.** Global Evaluation is done to ensure that the model of the instance architecture as a whole is well formed. For most parts of the evaluation, we can readily rely on the standard set of UML well formedness rules that are enforced by many modeling tools. The only other rule that seems to be applicable is to check if the model has been transformed into a set of two or more disjoint groups of model elements. For example, if a subsystem with a number of cooperating classes is only connected to the rest of the reference architecture by means of a single optional association, then it might happen that this subsystem is isolated after instantiation. Another example can be found in Figure 2 Statechart Diagram with Variability: Whenever one of the alternative optional transitions is removed to generate an instance architecture, the state it leads to remains present in the instance although it cannot be reached any more.

Global evaluation would be useful to detect these modeling flaws and would contribute to avoiding "dead code" in a system, that is, code that will never be executed because it will never be called from the rest of the system. Additionally, the evaluation could provide suggestions for more appropriate modeling (e.g., the whole subsystem should have been modeled as being optional, so that it would have been deleted during instantiation).

Although this kind of evaluation would be desirable, it was not yet investigated.

### 3 References

- [1] P. G. Basset, *Framing Software Reuse*, Prentice-Hall ISBN 0-13-327859-X, 1996

References

# Document Information

Title: System Family Architecture  
Description Using the UML

Date: 15 December 2000

Report: IESE-092.00/E

Status: Final

Distribution: Public

Copyright 2000, Fraunhofer IESE.  
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means including, without limitation, photocopying, recording, or otherwise, without the prior written permission of the publisher. Written permission is not needed if this publication is distributed for non-commercial purposes.