Modeling Variability -
From Direct Modeling to Generative Modeling

Øystein Haugen and Birger Møller-Pedersen
Department of Informatics, University of Oslo
{oysteinh | birger} @ifi.uio.no

Abstract

The paper shows how mechanisms of existing modeling languages (exemplified by UML 2.0) support the *direct* modeling of variability in software product lines/system families, and identifies where *generative* modeling (similar to generative programming) should be applied. Existing mechanisms are not only well-known mechanisms like composition, specialization, and generics/templates, but also less known (but in fact existing) mechanisms like subsetting and constraining parts of a system. Such mechanisms are useful for configuring specific system models based upon a system family model. Generative modeling turns out to be useful in cases where there are several orthogonal dimensions of variability.

1. Introduction

The main challenge when modeling a product line is to select the most appropriate mechanisms to describe variability.

When using UML1.x [9] for this purpose, there have been two approaches: either by *direct modeling* of variations by means of standard language mechanisms like e.g. composition, specialization, templates, or by *generative modeling* where a generation process produces specific models from a system family model, where model elements subject to variation are marked by means of extensions to UML in terms of stereotypes, see e.g. [7].

Direct modeling is illustrated in Figure 1. Composition may be plugging together components (from a library of component variants). Alternatively a framework capturing the common properties of the system family may be specialized.
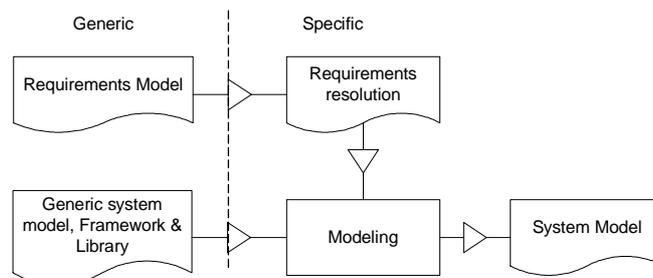

Figure 1 Direct Modeling

Generative modeling is illustrated in Figure 2. Approaches that rely on extensions of modeling languages marking model elements as variation elements typically model only the system family, with all variations included. Specific system models are generated on the basis of this system family model and a feature selection.
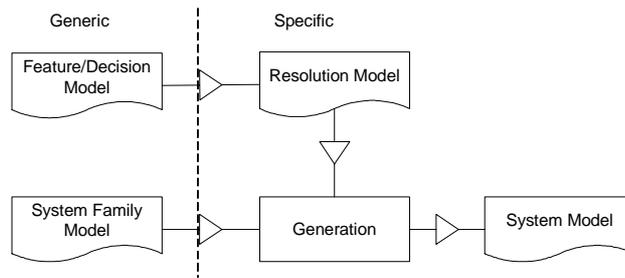
Figure 2 Generative Modeling

Generative modeling is a generalization of the notion of generative programming [3]. We have surveyed existing approaches and investigated the implications of applying UML 2.0 [10] to the modeling of product lines, see [2]. We found that UML 2.0 mechanisms like ports with well-defined interfaces, specialization (with redefinition of types) and templates (with constraints on type parameters) can be used for direct modeling of variations.

This paper gives a classification of kinds of variation, illustrating these by a set of variation challenges based on an example access control system. We analyze how UML 2.0 may handle such challenges, and where generative modeling mechanisms have to be employed. We discuss an approach to direct modeling of product line that is not properly covered by UML 2.0, *configuration* ([8]). In a configuration there are no explicit variation elements, but the specific systems are characterized by different configurations on structural properties of the general system family and by different values on attributes of parts of the system.

Section 2 introduces the example. Section 3 presents a classification of kinds of variations and illustrates these by a set of challenges. Section 4 tells how existing mechanisms (including configuration) can cope with some of the challenges, and identifies that for some of the challenges there is a need for mechanisms other than language mechanisms. Section 5 presents the difference between variation modeling and generative modeling.

2. The Access Control System example

We shall use an example (an access control system) to illustrate the use of mechanisms for variation modeling.
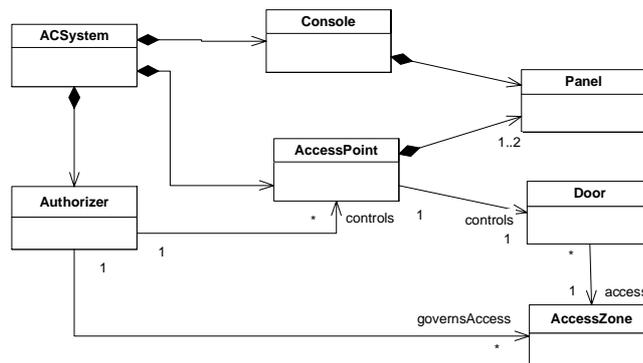

Figure 3 Domain model for access control

Access control has to do with controlling the access of users to a set of access zones. Only a user with known identity and correct access rights shall be allowed to enter into an access zone. Access control systems will provide services like

Figure 3 is a simple domain class model for access control systems: An AccessPoint controls access to a Door, through which the user enters the AccessZone. The

Authorizer controls access through an AccessPoint, and thus governs access to each AccessZone. Users interact with a Panel of the AccessPoint or the Console. There may be a Panel on either side of the Door.

A system family design model for access control systems typically include the modeling of the common architecture of all systems in the family and the modeling of variations on parts of this architecture.

Figure 4 illustrates how a class (ACSystem) defines the common architecture of all access control systems: Each access control system contains a number of access points (from 2 to 100), represented by the part 'ap' typed by the class AccessPoint. The access points interact with the users (via the port 'User') who request access and are granted or denied access to the access zones, and access points control the doors via the port 'Door'. The access points communicate with objects of type Authorizer to verify the validity of an access request. The ACSystem further has a Console that allows a supervisor to interact with the system (via the port Supervisor). Ports and parts are connected by means of connectors, and these specify potential communication. For the purpose of this presentation we have not included the full definitions of the ports. A complete model will define provided and required interfaces for each port.
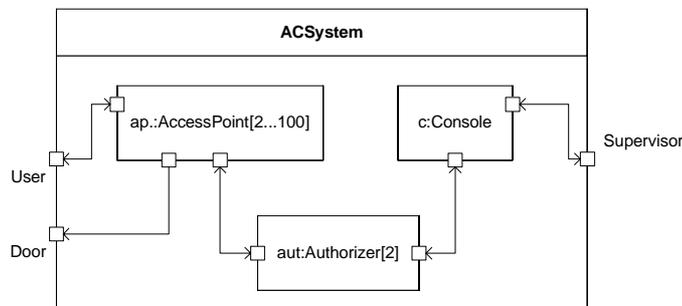

Figure 4 Composite structure of the access control system class

Any object of class AccessPoint will have two integer attributes, one for the floor number and one for the security level. The floor number should always be between 0 and 10, and the security level between 1 and 4, see Figure 5.
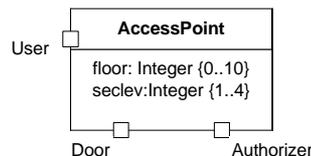

Figure 5 Constraints on AccessPoint objects

3. Variations, and how to classify them

In the following we classify variations and illustrate them by modeling a set of challenges to the example system.

In this classification of kinds of variation we have restricted ourselves to variations in the structure of systems, but we intend to pursue this classification also for other characteristics of systems. We assume that the structure of systems will be built from parts, where each part may have a number of ports (with provided and required interfaces) and where parts are connected by connectors. These terms are UML 2.0 terms, but most languages that are intended for the specification of the structure of systems (e.g. ADLs) have these kinds of constructs.

3.1 Part type specialization

This kind of variation narrows the types of system parts. This kind of variation assumes models of systems, where the parts are typed. In the example (see Figure 4), the system consists of three parts typed by AccessPoint, Console and Authorizer respectively. These types are e.g. defined as classes.

> *Challenge no. 1.* Systems may have one of two types of access points: Blocking access points are access points where an operator may block/unblock the access points, while logging access points are access points, that log what is going on by sending signals to a logging device.

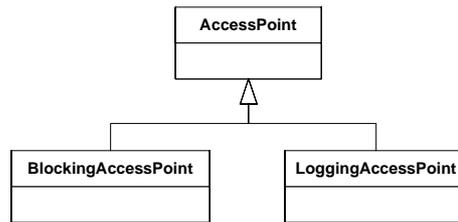The variations in types of access points are represented by two subclasses of AccessPoint, see Figure 6.

Figure 6 Different kinds of access points

3.2 Subsets of parts with property constraints

This kind of variation defines and names subsets of parts (that are sets in themselves) such that these subsets apply and thereby provide more accurate configuration. In our example we have two parts that are sets with more than one object: 'ap' and 'aut'.

> *Challenge no. 2.* The number of access points on each floor differs.

Property constraints are value constraints qualified by architectural properties. We illustrate this with three challenges:

> *Challenge no. 3.* All the access points on the same floor shall have the same security level.

> *Challenge no. 4.* On the ground floor the security level shall be high and the access points must be LoggingAccessPoints.

> *Challenge no. 5.* On all other floors the security level shall lower, and the access points shall be BlockingAccessPoints.

3.3 Configurations

Configurations define structure of subsets that still adhere to the original more general architecture.

> *Challenge no. 6.* The access points on the ground floor must be connected to one specific authorizer, while the other access points must be connected to another authorizer. The reason for this is due to the difference in security level. Furthermore on the top floors the access points may have card readers that may be either sliders or suckers.

This may also be called architectural variation.

3.4 Global dependencies

So far we have looked into variations that have to do with individual system elements (subsetting and values properties), or variations on the architecture of systems. All of these variations are associated with model elements that have a direct representation by a language construct (parts, subsets, properties of parts, ports/connectors).

There are, however, also variations that affect the whole system. A simple example is on the presence (or not) of system elements, e.g. whether a given element is part of specific systems or not (optionality). Another (more challenging) example is global requirements that will have a number of effects on a number of system elements.

In this example we only consider two simple cases of this kind of variation:

> *Challenge no. 7.* A special kind of systems does not have any console, and therefore access points of these cannot be blocking access points

> *Challenge no. 8.* Some systems must have encrypted communication on all channels between physically distributed units.

4. Variations, and how to model them

4.1 Part type specialization

There are a number of language mechanisms that answer the *Challenge no. 1*:
- plug-ins
- frameworks
- templates

4.1.1 Plug-ins

The plug-in approach (one example is [1], but also most ADL-based approaches, e.g. [4]) is based upon the idea of isolating the variations in components external to the stable parts of the system, with well-defined interfaces that apply to all variant components.

This approach applied to the access control system may at first appear awkward, see Figure 7 a). However, the model expresses that all access control systems have two authorizers and one console. The variation is what kind of access point they have, and this is modeled by a port to which any of these different kinds of access points may be connected.

a)   b)
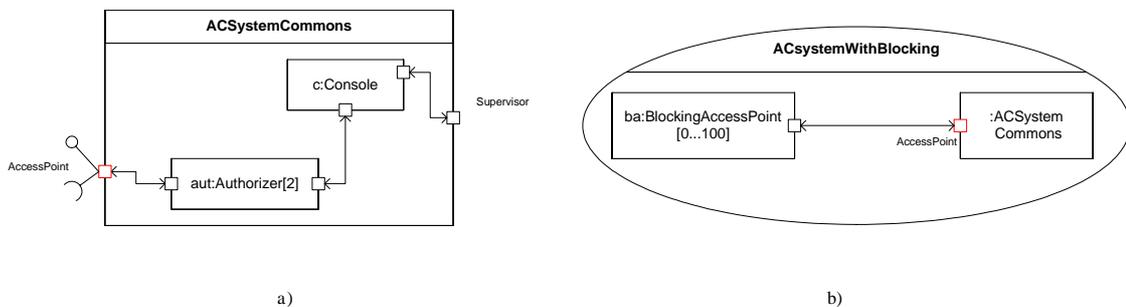
Figure 7 a) Common part of access control systems   b) Plugging in BlockingAccessPoint

Figure 7 b) specifies a system with blocking access points, simply by connecting a set of BlockingAccessPoint objects to the port of the system.

The plug-in approach is directly supported by composite classes/collaborations in UML 2.0. Plug-ins are modeled by parts (of a composite class or collaboration representing the whole system), connection points for plug-ins are modeled by ports, and plug-ins are connected by connectors. Ports in UML 2.0 have both provided and required interfaces.

4.1.2 Frameworks

Frameworks are designs where the structure and default behavior of systems are defined and where certain so-called hotspots represent variations. These are the only parts of the model/program that may be defined differently for specific systems – the rest of the architecture and behavior is common to all systems made according to the framework. The idea of frameworks started out in programming [5]. While [6] provides mechanisms for the modeling of frameworks by means of stereotypes (e.g. indicating what are the hotspots), the following shows that even existing mechanisms provide hotspots.

By defining the class AccessPoint locally to the ACSystem (Figure 8 a), UML 2.0 provides the means for redefinition of this class in subclasses of ACSystem. The class AccessPoint is thereby a hotspot.
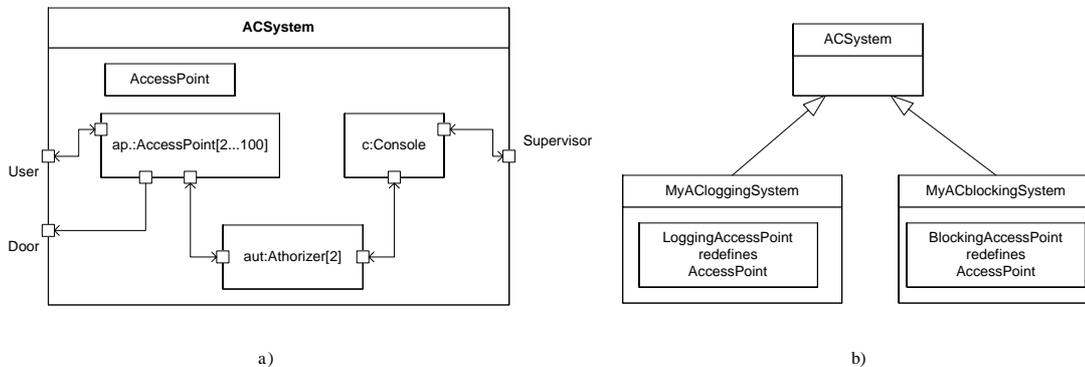

Figure 8 Framework with local class AccessPoint being redefined

In Figure 8 b) we have illustrated this by defining two different ACSystems based upon the ACSystem framework. Redefining the class AccessPoint is sufficient to define a variation. The class ACSystem in Figure 8 a) defines a framework. The whole composite structure of ACSystem is inherited; the only difference is that the type the 'ap' part of this structure is redefined to a more specific type.

4.1.3 Templates

An alternative to framework with redefinable classes is to define ACSystem with type parameters for those types that have to vary. In Figure 9 a) the type of the set 'ap' is not AccessPoint, but a class parameter 'apType'. A specific ACSystem can be derived by binding 'apType' to e.g. BlockingAccessPoint, see Figure 9 b).
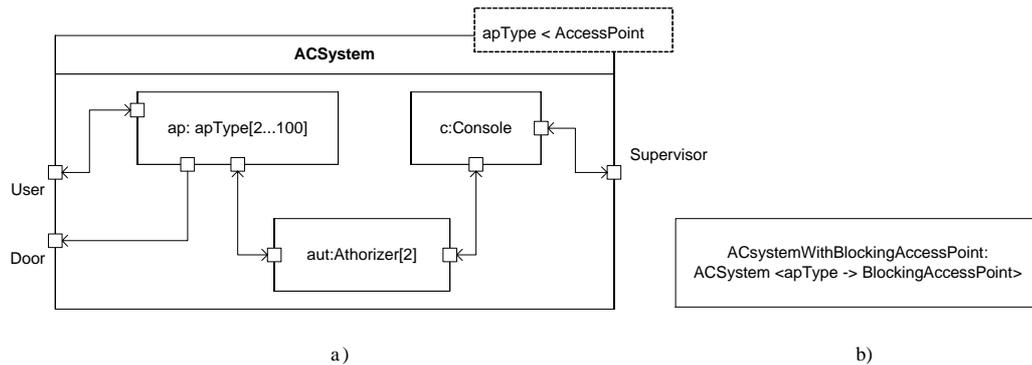
Figure 9 ACSystem with a template parameter representing the type of access points

The specification of 'apType' in Figure 9 includes a constraint ('< AccesPoint'). This implies that it is only possible to bind 'apType' to classes that are subclasses of class AccesPoint, thereby ensuring at least the properties of AccesPoint.

4.2 Subsets of parts with property constraints

When it comes to the *Challenge no 2*, *3*, *4* and *5*, the challenges of property constraints, neither of the mechanisms described in 4.1 can be used. The mechanisms described so far can only redefine types and thereby affecting all elements of these types. For *Challenge no 2*, *3*, *4* and *5* we will have to have mechanisms that can model different subsets of a given set ('ap').

There are means in UML to express that elements of a given set belong to different subsets. The set 'ap' of AccessPoints may contain objects of LoggingAccessPoint and of BlockingAccessPoint or other subclasses, and the way to express it in UML 2.0 is through special subset constraints.

Subset-constraints were introduced in UML 2.0 to cope with the need to express association specialization [12]. Such subsetting on association ends can be found numerous places in the UML 2.0 metamodel. This can help us part of the way to express what we want for our particular access control system where the set 'ap' of AccessPoints has been split into one set of BlockingAccessPoints for the top floors and one set of LoggingAccessPoints for the ground floor.
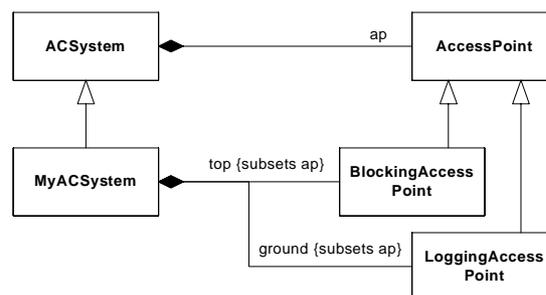

Figure 10 Subsetting association ends

Figure 10 shows a legal construction in UML 2.0: subsetting of association ends.

Our answer to the challenge is to apply subsetting to parts. This is possible because UML 2.0 have subsetting not only for association ends, but for properties in general, including parts. We can thereby express that the set 'ap' of AccessPoints shall be seen as a number of subsets. These subsets must have a clear relation to the original 'ap' set, but shall also describe individual peculiarities. In Figure 11 this is illustrated by a notation where the names of the subsets following the syntax for roles: <subset name>'/'<set name>.
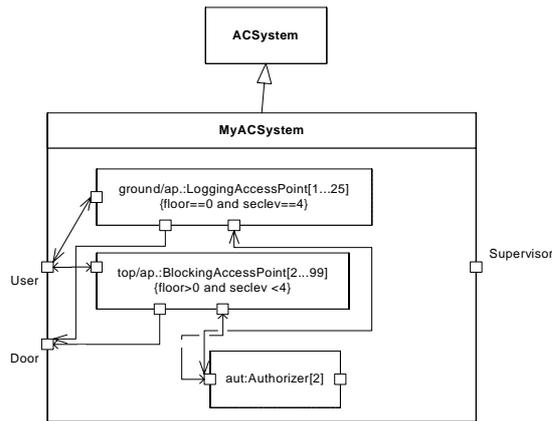
Figure 11 Configuration by subsets

4.3 Configurations

We have now shown that the subsets can have different specifications and thereby fulfill the *Challenges 2* to *5*. *Challenge no. 6* is concerned with the architecture of the system. We want to express that the ground set of access points have different connections than the top set of access points. It is actually straight forward to achieve this also since we have two distinct descriptive elements that may have different connections. We must make sure that even though the connections are different, they must still satisfy the general description of the ACSystem (that is how 'ap' is connected in ACSystem).
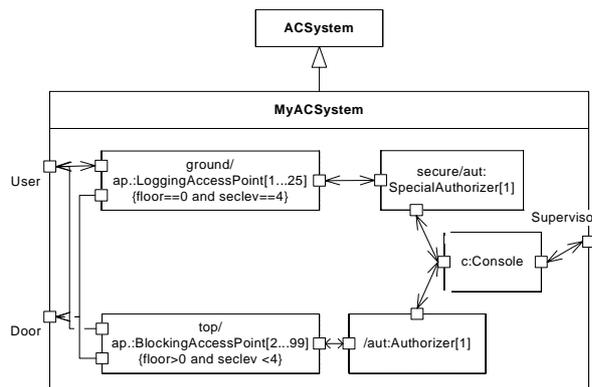

Figure 12 A specialized composite architecture

In Figure 12 we have distinguished between the ground floor access points and the top floor access points and made the connections to different authorizers depend on these subsets.

There are ADLs (e.g. [4]) that in addition to concepts like component, connector, interfaces supports the notion of configuration, in [4]defined as the topological arrangements of components and connectors as realized by links. Figure 12 shows that this may be obtained by the existing subsetting mechanism.

Each access point has a card reader. In *Challenge no. 6* we also want to express properties of the card readers depending on their surroundings. On the first floor we want only slider readers, while on the second top floor we may have both sliders and suckers. A sucker keeps the card while validating, while a slider does not. We may express this directly as in Figure 13.
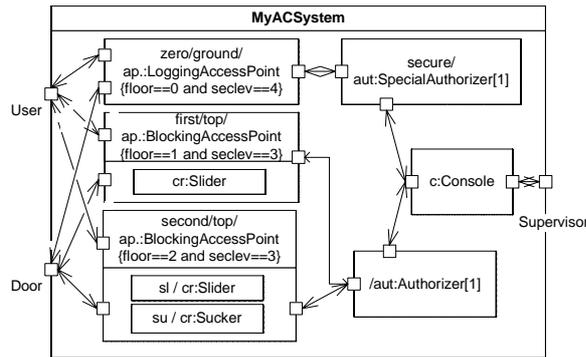
Figure 13 Nested configurations

Here we have deviated even more from the current UML 2 notation, and by doing so making the elaborate diagram reasonably transparent.

4.4 Global dependencies

The configuration shown in Figure 13 may be transformed into proper UML by introducing new classes for the different access points that have differences inside. UML. By doing so the description is in a standard language, but not as transparent.

Our *Challenge no. 7* poses yet another problem that is not so easily solved in UML 2.0 (or in any modeling language). The challenge requires the satisfaction of a constraint which combines elements from different elements of the model and that has no specific construct in UML 2.0.
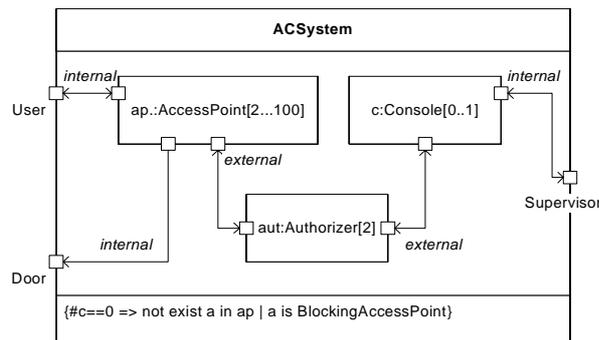

Figure 14 ACSystem basis for generation

In Figure 14 we show a new version of the general ACSystem description quite similar to the one in Figure 4, but as we shall see there are some differences. To answer *Challenge no. 7* we have changed the 'c:Console' to have multiplicity [0..1] and we have added an elaborate global constraint that is intended to make sure that systems without consoles cannot have BlockingAccessPoints. The constraint could possibly have been formulated in proper OCL, but it would hardly be more readable by that. If the constraint had been a constraint on the model (and not the system), then the multiplicity would not have changed, but there would rather be an annotation (e.g. 'optional') on the 'c' part. In order to get a model of a system with no console, a new model would have to be generated, where the Console part (and whatever is connected to it) is removed, while a model of a system with a console had been generated by simply removing the 'optional' annotation.

To answer *Challenge no. 8* we have to introduce special markers on the connectors, either "internal" or "external" indicating the kind of communication. External connectors are between units that are physically distributed, while the internal connectors are for communication where the distance is so short and the environment so

secure that no further security measures needs to be taken. The reason for these special markers is that in order to make a model of a system with encrypted communication, we will have to take the (annotated) model in Figure 14 together with a model for how encryption is obtained and generate a new model.

By introducing these channel annotations we have actually departed slightly from UML. We could have applied stereotypes, but chose not to in order to show that when there is an intended generation phase, the starting point is often a model in a slight augmentation of the standard modeling language.

We now add an additional model representing the encryption (Figure 15).
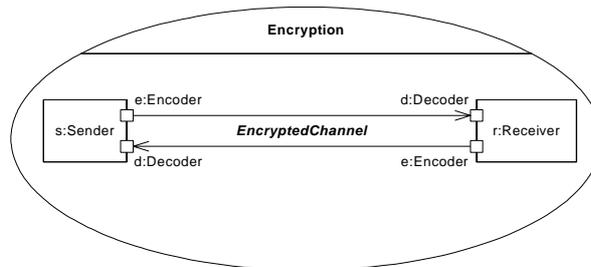

Figure 15 Encryption model

Again we did not need to use UML, but there are benefits from using a homogeneous set of models. Given this encryption model, and the extra information that Encoder and Decoder shall be realized by the port types E06 and D06 respectively, a generation could then produce the system model in Figure 16.
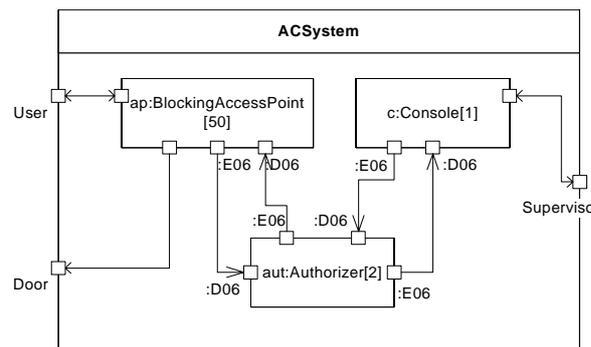

Figure 16 The generated system model

As illustrated by Figure 16 the resulting generated system model may be rather elaborate, while the sources of the generation may be simple and transparent.

5. Modeling versus generation

In the introduction we introduced the two different approaches to variability: direct modeling and generative modeling. While direct modeling exploits the standard language concepts, generative modeling means that a new model (or program) is produced from an original set of models (or programs).

It is obvious that the generative approach can be used to express whatever the non-generative approach can express. The models we made for the challenges 1 to 6 could as well have been generated. The use of feature models and decision/resolution models are examples of approaches that rely on this to the extent that all variabilities are captured only by feature models and associated decision/resolution models.

Although it is tempting to capture all kinds of variability by special kinds of (feature/ decision/ resolution) models, the danger is that the existing mechanisms (like composition, generics, etc.) have to be re-invented as part of the languages used for

making these special models. On the other hand, if existing language mechanisms shall be used for what they can be used for, it is important to know what the limits to non-generative approaches are.

In recent years support for cross-cutting concerns in the form of aspects has been provided through mechanisms that according to the above distinction belong to generative programming. This does not mean that it is impossible to make language constructs such that aspects may be interpreted directly, but the available mechanisms in programming (Java) and modeling (UML) have not been considered aspect-oriented enough. Therefore we have seen many generative approaches to aspect orientation.

We have shown that generative modeling is closely related to the discrepancy between the modeling needs and the available mechanisms. An extra generation step closes this gap. The last challenges could not be modeled by means of existing mechanisms. The most extreme approach in this respect is that of Domain Specific Languages, where the designers in principle claim that there are no available mechanisms at all, such that the gap is actually the modeling of the whole domain and the definition of a language to capture this. In practice this is of course not true since Domain Specific Languages tend to be very similar to earlier attempts and quite similar to general-purpose languages.

On the other side of the spectrum of generative approaches, we have the simple model annotations. We know them from program languages where they were called compiler directives or pragmas. These were hints in the program text that helped the compiler produce the most effective code. The pragmas of programming languages developed into full fletched macro languages such as the C macro language. The preprocessor actually produces a pure C program that is then fed into the pure C compiler. In UML we have had stereotypes for exactly the same purpose. Common to the annotation approaches is that their effects are normally very local. Simple, local substitutions most often do the trick.

In between these extremes, we have seen compromises. In both [11] and [2] the whole model is divided in three – the base model, the variation model and the resolution model.
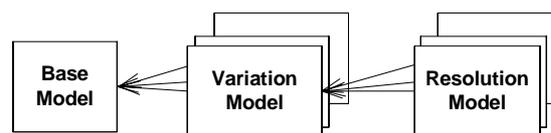

Figure 17 Base-Variation-Resolution

Typically the base model is not an adequate model in the modeling language, but it is rather a collection of fragments in the modeling language. One base model may have more than one variation models. Resolution models in turn apply to variation models. The benefit of these approaches is that existing languages mechanisms for variability can be exploited in the base model.

6. Conclusion

Figure 18 summarizes the classification of variations that has been presented, spanning from variation of types of system elements to variations that require model generation techniques. It has been illustrated that some of these variations may be covered by direct modeling using existing language mechanisms, and it has been demonstrated that there are some kinds of variation that cannot be covered by modeling, not even by specifying constraints (e.g. in terms of OCL), but has to be captured by generative modeling.
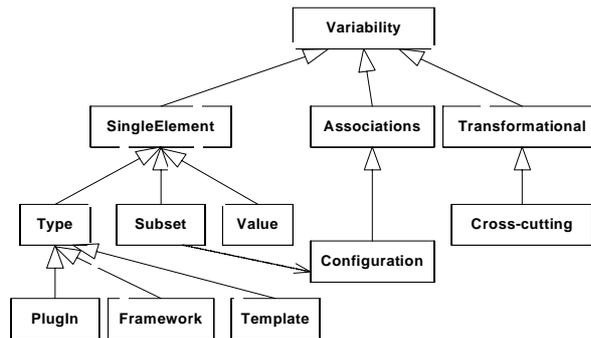
Figure 18 Variability Classification Model

The classification is based upon the use of UML for variability modeling, and it is based upon the modeling of system families by means of either composite classes or collaborations. However, it covers the way most Architecture Description Languages model systems.

We believe that given a modeling language to be used for variability modeling, it is useful to apply this classification in order to avoid generative modeling where it is not needed, and to apply it where it is really needed.

7. References

[1]   America, P., Obbink, H., Ommering, R. v., and Linden, F. v. d.: *A Component-Oriented Platform Architecting Method Family for Product Family Engineering*, SPLC1, Denver (USA), 2000, 576, Kluwer International Series in Engineering and Computer Science.

[2]   Bayer, J., Gerard, S., Haugen, Ø., Mansell, J., Møller-Pedersen, B., Oldevik, J., Tessier, P., Thibault, J.-P., and Widen, T.: *Consolidated Product Line Variability Modeling*, in *Research Issues in Software Product-Lines*, Käkölä, T., Ed., Springer, 2006.

[3]   Czarnecki, K. and Eisenecker, U.: *Generative Programming: Methods, Tools, and Applications*, vol. 864: Addison-Wesley, 2000.

[4]   Dashofy, E. M. and Hoek, A. v. d.: *Representing Product Family Architectures in an Extensible Architecture Description Language*, International Workshop on Product Family Engineering (PFE-4), Bilbao, Spain, 2001,

[5]   Fayad, M. and Schmidt, D.: *Object-Oriented Application Frameworks*, Communications of the ACM, pp. 32-38, 1997.

[6]   Fontoura, M., Pree, W., and Rumpe, B.: *The UML Profile for Framework Architectures*: Addison-Wesley, 2001.

[7]   Gomaa, H.: *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*, vol. 736: Addison-Wesley, 2004.

[8]   Haugen, Ø. and Møller-Pedersen, B.: *Configurations by UML*, 3rd European Workshop on Software Architecture - EWSA 2006, Nantes, France, 2006, to be published in Lecture Notes in Computer Science, Springer Verlag.

[9]   OMG: *OMG Unified Modeling Language 1.4*, Object Management Group 2000.

[10]   OMG: *OMG. Unified Modeling Language 2.0*, OMG. ptc/2004-10-02 2004.

[11]   Pohl, K., Bökle, G., and Linden, F. v. d.: *Software Product Line Engineering - Foundations, Principles and Techniques.*: Springer, 2005.

[12]   Rumbaugh, J., Jacobson, I., and Booch, G.: *Unified Modeling Language Reference Manual*, vol. 736, 2 ed: Pearson Education., 2004.