Chapter 6
Variability Realization Techniques and Product Derivation

Rafael Capilla

What you will learn in this chapter

- *The notion of variability realization and product derivation.*
- *The relationship between binding time and product derivation.*
- *Automated product derivation approaches.*

1 Introduction

One of the ultimate goals of the usage of variability techniques is to allow the configuration of the software products under the product line approach. As different binding times are possible, different variability implementation mechanisms can be used to realize the variability at different stages in the software development lifecycle. Once variability is defined in the architecture and implemented in code, products can be configured at the end of the product line or even reconfigured at runtime. Hence, the variability defined in the architecture can be instantiated for configuring the product portfolio at different stages (e.g., pre-deployment, end of SPL, installation, runtime). Besides, variability realization techniques are intimately linked to the way and the moment products can be deployed, and several alternatives can be chosen to select the best configuration and deployment strategy. In this chapter, we will learn about variability realization techniques.

R. Capilla (✉)
Rey Juan Carlos University, Móstoles, Madrid, Spain
e-mail: rafael.capilla@urjc.es

R. Capilla et al. (eds.), *Systems and Software Variability Management*,
DOI 10.1007/978-3-642-36583-6_6,

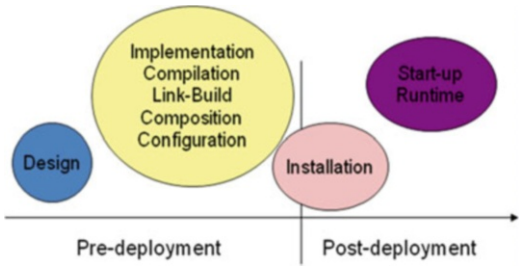

Fig. 6.1 Variability realization stages before and after deployment time

2 Variability Realization

In the solution space, the provide variability is realized by instantiating their variants and variation points in order to configure the products with the right and allowed values. Therefore, the realization of the software products implies to know at a certain time in the software development process which will be the values of the configurable options defined in the architecture and implemented in the core assets and products as well. Variability realization is intimately linked to product derivation, aimed to produce the concrete products once the values of the variants and variation points are known.

Definition 6.1. Variability realization technique
It is the way in which the variants of any family member are realized using a particular variability implementation technique at a given binding time.

The realization of concrete software products implies that the variable interfaces between components must be known, in addition to the invariants described in the architecture. The realization of the variability through the interfaces that may vary is crucial to set the right links between software components, as these interfaces act as a selector of the right component when more than one alternative exist. In addition, the realization of the variability must check the compatibility of the constraint rules, hundreds in commercial software, among the variants selected to avoid incompatibilities during the product derivation.

Definition 6.2. Product derivation
It is a stage in the software product line life cycle where software products become the resultant of a selection and configuration process of the variable design options defined in a variability model.

The software engineer must decide when to realize the variable options, and the flexibility provided by the existence of different binding times offers software engineers a way to delay their design decisions to a later stage. In Fig. 6.1, we organize the different product realization stages based on the moment in which products are or will be deployed.

We have to mention that installation time is not a real post-deployment variability realization stage as it is somehow in the middle, but we preferred to classify the realization of the variability during product installation closer to post-deployment time.

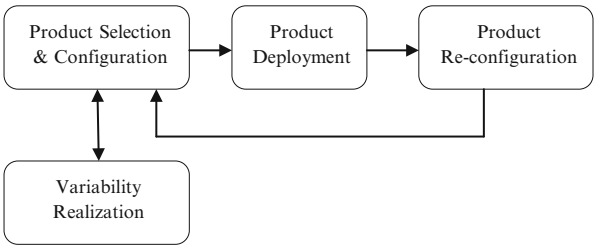

Fig. 6.2 Product derivation activities. The runtime reconfiguration of variants may lead to the selection of new variants and variation points and, in some cases, to a product redeployment phase

An exhaustive taxonomy of variability realization techniques and the factors that are relevant to implement variability can be found in [1], but the current trend in software development for several application domains like self-adaptive systems and service-based system pushes the realization of the variability to runtime modes. In this chapter, we will distinguish three major development stages in which we can realize the variability and according to most common binding times [2].

2.1 Product Derivation Activities

The ultimate goal of a product derivation process, as part of the SPL application engineering lifecycle, is to produce a configurable or configured software product. However, product configuration can be enacted at the beginning of the derivation process at early binding times, or it can be also executed at a very late stage if a product has to be reconfigured once deployed. Configuration is sometimes done to select the variable options that will be included in a product before the variability is realized to concrete values, while in other cases, a reconfiguration process happens at the end of the product line or during system execution. Moreover, product configuration and variability realization can also overlap at the same binding time if we realize the variants at the same time these are selected. At the end of the derivation process, products are installed and deployed in the physical nodes of the system.

As a summary, we show in Fig. 6.2 how these concepts are related and based on the binding times where these activities happen. Initially, product configuration starts by selecting the variable options that will be included in the product, and this activity may happen at different binding times, in which the realization of the variability will take place immediately after. Once the variable options match to concrete values, the executables can be deployed. However, post-configuration operations can be possible when the systems need to be reconfigured at post-deployment time, and dynamic variability plays an important role for systems that require runtime adaptation.

Figure 6.2 describes the major activities of a generic product derivation process. Once the input requirements define the selection of the variants of a new product, a product selection and configuration process chooses the right variants for configuration purposes, and variants are realized according to a particular implementation technique and the allowed values for those variants. Once the variability is realized and the product already configured, installed, and deployed, any post-deployment

activity or runtime reconfiguration of variants may lead to a new selection and configuration of the variable options. In that case, the reconfigured product or the new product (i.e., a different selection of the variable options can lead to different products) can or must be deployed again, while in other situations, no new deployment is required (e.g., the case of dynamic variability used to, for instance, activate a feature at runtime). The figure does not show testing activities that should be carried out to validate the selected product configuration.

In addition to Fig. 6.2, we detail in Table 6.1 which tasks encompass each of product derivation activities. For each of the major activities of Fig. 6.2, we provide the subtasks that are commonly needed and the most suitable binding times under which these tasks may happen.

2.2 Realization at Design Time

At design time, the realization of all variants and variation points is made at the architecture level. The variants in the design are manually operated, as the variability is considered statically in nature. Standard notations like UML offers few mechanisms (e.g., stereotypes, tagged values) to describe the variability of a feature model in the architecture, and the logical formulas describing relationships between variants do not have a direct correspondence in UML diagrams and they must be represented using a different notation or language. Therefore, the steps to realize the variability at design time are:

(a) Selection of variants and variations points defined in the architecture.
(b) Selection of allowed values.
(c) Depiction of the product architecture by instantiating the variants with appropriate values for each single product.

In Fig. 6.3, we show an example of a UML diagram that belongs to the software architecture of system X (left side of the figure) containing five variants and two variation points. At design time, the software engineer selects the variants to realize the construction of system X.1, and he/she derives the product architecture for that system. In this case, variant 3 and variant 4 with their corresponding values have been selected. Variation points and variants are selected and instantiated also for the product architecture.

2.3 Pre-deployment Realization

Products can be configured in the customer site and afterwards installed and deployed in the client side. When the variability of products is realized in the customer site, the variants and variation points can be instantiated at different binding times, depending on how the product is built and configured. At implementation time, the variability can be implemented in variables and the alternatives and

Table 6.1 Tasks encompassing product derivation activities

Activity	Tasks	Binding time	Description
Product derivation	Product configuration Variability realization Product deployment Product installation Product reconfiguration	From design to runtime	Product derivation comprises the main four activities described in Fig. 6.1 in order to build a software product. Product derivation is intimately related to variability realization techniques
Product configuration	Selects the variants to be included in the product Configure the allowed values Check dependency rules	Configuration time Pre-deployment Post-deployment Start-up	Product configuration deals with the selection of the variable options at different binding times. Excludes and requires rules must be checked
Variability realization	Instantiate the values according to the variability implementation techniques used Check constraint rules	From design to runtime	The variants match to concrete and allowed values. Rules delimiting the scope of products must be run
Product deployment	Product installation Node selection	Installation and deployment times, as no further binding time is considered	Installation comprises the nodes where the system functionality will be deployed and maybe some post-configuration activities
Product reconfiguration	Product configuration Variability realization Product deployment	From design to runtime	Reconfiguration procedures may lead to a feedback from runtime to variability selection and realization, and product redeployment activities as well

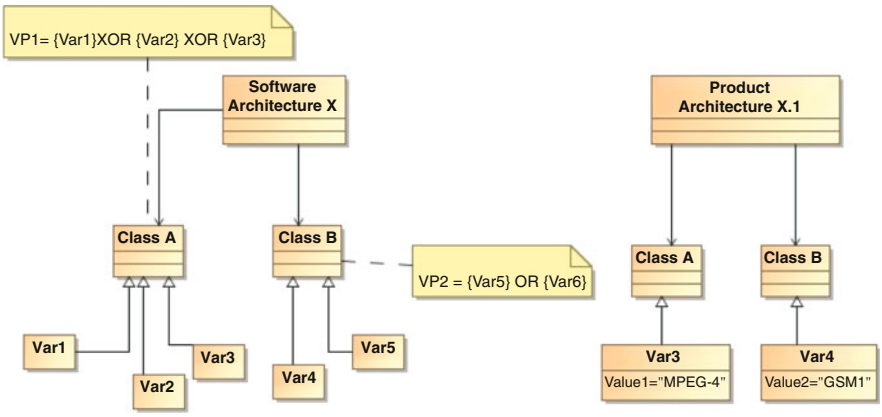

Fig. 6.3 Variability realization at design time

constraints described are often described as *if-then-else* constructions or using constraint programming. The realization of the variability depends on how this is implemented. For instance, the realization of the variants can be done statically in the code changes or more dynamically using dynamic libraries containing the configurable options. If we use an object-oriented approach, variability can be implemented using inheritance to separate the common functionality in superclasses from other variable option defined in subclasses which can be instantiated during the derivation process.

Example 6.1. Variability specialization through OO inheritance

The 3D scene of a virtual reality (VR) system is composed by 3D objects that constitute a hierarchy where objects are successively decomposed in polygons starting from a root object or node. Because the 3D database contains several megabytes and the time for loading the 3D scene during first start-up can delay several minutes, the way in which this hierarchy is organized at the architecture and implementation is critical, as some objects may appear initially hidden or might be unnecessary to show all the details of some of these 3D objects. Therefore, arranging and organizing this hierarchy in a particular form is quite important to reduce the start-up time. In this example, we used a particular object hierarchy (tested using simulation) to reduce the start-up time of the 3D scene and the variability techniques based on inheritance were used to group objects with common behavior [3].

In addition, at compilation and link or build times, directives expanding program macros, variables and compiler flags {e.g., `#ifdef, #include, #define`}, and Makefiles linking the program modules in a specific order are instantiated to produce different configurations or versions of the same product. For instance, a compilation variable can be used to discriminate a stand-alone version from a distributed one or add a security module not present in a different release. Makefiles use variables to make more flexible link and build options when generating the binaries, such as shown in the following code are certain flags that are stored in variables:

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
        $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
        $(CC) $(CFLAGS) $< -o $@
```

Moreover, the idea of staged variability fosters the composition of features that can be added or removed to derive different product configurations (e.g., a scientific calculator has several versions of the same products and the product line approach used composed new functionality by selecting new variants). The software engineer selects and deselects features of each new version of the product. Hence, stage configuration becomes an important process applied in an SPL for configuring software products and where people make the right configuration choices at different stages. As described in [4], staged configuration of feature models constitutes a stepwise refinement of the variability model.

In some cases, this refinement leads to a specialization where groups of features are selected during the product configuration process and yields a specialized feature model. Specialization can be seen as a subset of the overall set of configurations and often done via transformations. In this context, baseline architectures play an important role for specialization and derivation processes as new product releases are yield as a result of successive stepwise refinement by adding and removing features from the baselines or from a concrete product configuration. Then, extensibility of the architecture becomes crucial to synthesize different product configurations or release products for different platforms.

2.4 Post-deployment Realization

The configuration of the variability and product realization in the customer site (post-deployment activities) often involves installation and post-deployment procedures where products are configured and deployed on behalf of a set of configurable options (e.g., parameters) that tailor the product to a specific environment or user preferences.

More dynamically, products may change the configuration of their variable options during start-up (e.g., first start-up or on every start-up) time as a system operator can configure certain variable options. For instance, the installation of a

new version of an operating system allows users to configure certain parameters of the target machine (e.g., language, screen resolution) or to select between two different preinstalled versions. On every start-up, the variability can be stored in configuration files (e.g., XML files) or local databases that are uploaded dynamically (e.g., a new user profile that has assigned new privileges). Other situations may deal with the dynamic upload of software modules or libraries that affect to the system configuration, as in some cases, the system needs to be restarted.

Finally, during system execution, the selection of variants happens while the system is running. The ability to select a new variant or to activate/deactivate features is considered a pure runtime variability realization (e.g., an adaptive system that realizes a reconfiguration of certain design options) which often happens at post-deployment time.

As a brief summary, we have to mention that depending on the concrete binding time and on the implementation language selected, the variability realization technique would be different (e.g., parameterization, inheritance, dynamic libraries, user-configurable options, etc.), and runtime variability realization techniques require more complexity and implementation effort.

3 Automated Derivation and Runtime Reconfiguration

The automation of product derivation and configuration tasks is quite important for certain variability management and product derivation operations. As some systems deal with runtime concerns, automating product deployment is increasingly interesting for such systems that require unattended and autonomous manual operations.

Usually, product configuration is perceived as a manual activity but dynamic SPL approaches attempt to manage the automatic activation and configuration of system features or perform an automatic redeployment once the system has been reconfigured dynamically. Some systems with stringent requirements require strict runtime adaptation of their systems options, while in others, it can be a semiautomatic human-guided procedure (e.g., a pluggable smart home system able to plug new software modules automatically and the variability is configured manually afterwards before launching the new functionality).

The automation of product derivation processes can be achieved following a generative approach or a specific model-driven development (MDD) where models are transformed before the final variability realization mechanism realizes the design choices. The input for such automatic process is a feature model or UML model that requires some kind of transformation before the variants are selected. For instance, in [5], an SPL derivation approach, built on the top of Rational Rose RT, provides automated support for developing multiple SPL views in UML and using the feature model as the unifying view.

One important topic in today's automation techniques for product configuration is automatic deployment. As systems or part of them are installed and deployed periodically (e.g., due to the installation of critical updates or because a new hardware is plugged and a reconfiguration operation is needed to support the new

functionality), there is an increasing demand to provide some degree of automation. Therefore, automating configuration and derivation processes in conjunction with deployment activities facilitates the task of software engineers, as many system configuration and installation procedures could be enacted unattended and automatically. In this scenario, software variability can play a key role to handle a set of configurable options that can be managed in automatic mode at runtime.

Some authors [6] suggest a model-driven engineering approach using variability mechanisms under a product line context to automate the customization and deployment of software products. This approach advocates the use of transformation languages such as ATLAS Transformation Language (ATL) and Acceleo, which extends the capabilities of the GenArch[1] software product line tool in order to transform software processes based on the Eclipse Process Framework[2] (EPF) to jPDL workflow language specifications and enable the deployment and execution of such processes. A feature model is used to specify the variability of these software processes and a product derivation tool allows the selection of the relevant features from an existing process, enabling automatic derivation from the software process to a workflow specification. Model-to-model transformations (M2M) facilitate the translation from an EPF specification of an automatic customized process to jPDL elements. Such automatic procedures often exploit model-driven engineering techniques to realize the transformations from high-level models (e.g., a UML specification) to code assets. Another technique which can be used is generation, which realizes stepwise refinements from baselines.

Consequently, the automation of product derivation and configuration activities requires additional coding effort to support automatic management of the variable options, as these configurable choices are sometimes handled by an automatic procedure, while in other cases, the ultimate goal is to leave some of these design choices to be modified by the user at runtime and post-deployment time.

Reconfiguring products at runtime may require in some cases to restructure the entire or a subset of the variability model. Reorganizing the structural variability model at runtime is challenging and hard, but this topic is out of the scope of this chapter. However, other runtime reconfiguration operations may imply automatic activation and deactivation of certain system features in order to meet new context conditions. Any runtime reconfiguration demands automatic redeployment mechanisms to meet the runtime condition, as well as additional runtime checks (even if a system changes its operational mode for some time) to ensure that the new configuration is the right one and properly set. Autonomic computing, pervasive and context-aware systems, service-based systems, and self-*systems are the most suitable candidates for runtime reconfiguration operations supporting variability. Other systems demand reconfigurable operations when new modules are plugged and unplugged and dynamic libraries or software modules can be selected automatically or with minimal human intervention using variable and configurable options. In those more complex cases, policies for runtime changes must be used to manage

[1] http://www.teccomm.les.inf.puc-rio.br/genarch/

[2] http://www.eclipse.org/epf/

the different situations that might arise during the selection of different configurable options and to detect incompatible product configurations.

4 Areas of Practice

4.1 Tooling

Several tools and approaches have been developed to support SPL derivation activities. From a methodological point of view, the "ConIPF Variability Modeling Framework" (COVAMOF) derivation process [7] describes the practical realization of variability for product families through a set of steps that go from the feature model to the component implementation and each of these levels are associated to COVAMOF variability views which capture the dependencies and relationships of the variability model. COVAMOF uses XML-based feature models and *#ifdef* constructs to describe and mange the variability information. The COVAMOF derivation process first configures the product to bind the variations and then realizes the product on the SPL artifacts in order to make effective the values of the variants.

Cirilo et al. [8] compares how three SPL tools (i.e. CIDE, pure::variants, GenArch^{+}) use configuration knowledge to compose the product line variability to derive the SPL products. This knowledge, used in configurable product lines, defines the implementation and composition of the variability for product derivation tasks. The comprehension of this configuration knowledge is crucial to understand domain-specific abstractions which are used for modeling coarse-grained variability and describe the relationships between SPL variability and code assets, annotations in feature models, and fine-grained variability implemented in class attributes and methods.

4.2 Experiences

In several industrial experiences, configuration and variability realization processes become relevant for product derivation. One early experience in the automotive domain [9] enables product derivation through the selection of combined variants aimed to support the right product configuration.

The well-known Koala model for handling the diversity of software products in the consumer electronics domain [10] is a clear example where the size and complexity of software products increasingly growing required a robust variability model able to handle this diversity. The Koala model proposes a strict separation between component and configuration development, as component builders do not make assumptions about the configurations in which components will be used.

Each component provides its functionality through a well-defined set of interfaces (e.g., the signal of a TV tuner is fed by a high-end input processor (HIP) that decodes luminance and color signals which are the inputs to a high-end output processor (HOP). All these devices are controlled by software drivers using a serial IC2 bus, as each driver requires, and IC2 interface that must be bound to an IC2 service during system configuration. A configuration in Koala is a set of components connected to form a product. In Koala, static binding is used during compilation running at configuration time.

In addition, the Koalish modeling language extends Koala and used for automating the product individuals in configurable software product families (CSPFs) [11]. Koalish is built on Koala and adds new variation mechanisms for selecting and configuring the type of parts of components, including constraints for specific individuals. In Koalish, configurations are sets of component and interface instances, and the relations describing which component instances are part of other component instances. The authors introduce the notion of valid configuration as not all possible configurations represent a system. On this basis, the WeCoTin is a prototype configurator tool operating on the product configurator modeling language (PCML) in order to ease the configuration of software product lines and feature models [12]. Reinforcing previous proposals, other authors [13] describe an analysis of the derivation process in two software companies for configurable software product families, from requirements to product delivery.

Regarding automatic product derivation, an experience using multi-agent systems (MAS) under a product line approach is described in [14], where a model-based product derivation tool (GenArch) is proposed for use in the application engineering lifecycle. GenArch consists basically of three steps: (1) automatic models construction, (2) artifact synchronizations, and (3) product derivation, which comprises customization and composition of the SPL architecture.

5 Summary

Evolution is an important aspect for today's software systems, and software variability reduces the barrier for systems that have to evolve more dynamically. Hence, feature models must be ready to support the selection and unselection of features and configuration operations during product derivation and deployment activities.

In this chapter, we have discussed the characteristics of major variability realization and derivation activities. Product derivation tasks can be organized according to pre- and post-deployment binding times, as this separation of concerns is easier to understand when and where (i.e., developer and customer sites) products can be realized. In addition, the categorization of derivation activities becomes important to know which kind of subtasks and which binding times can be used in any derivation process using variability.

The utility to realize the derivation at different binding times will depend in many cases of the type of systems we want to build and deploy, as not all software systems may require to support runtime concerns.

The areas of practice described in the chapter are several and show representative types of systems and applications in various areas that exploit variability realization techniques in different ways and with different binding times, as some of them have different deployment and configuration requirements.

6 Outlook

No one doubts about the importance of product derivation and deployment activities for variability management. In this context, automating reconfiguration and redeployment activities for critical and real-time systems is crucial, as systems using context information are more and more frequent. Systems using variable options evolve much better in dynamic contexts compared to those others than use more rigid approaches.

Finally, regarding the variety of variability realization techniques, we did not want to describe detailed examples on how each variability realization technique can be implemented, as this depends on the language or platform used. Rather, we preferred to provide an overview of the most common techniques used, organized around the time in which variants can be bound.

References

1. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exp. **35**(8), 705–747 (2005)
2. Fritsch, C., Lehn, A., Strohm, T., Bosch, R.: Evaluating variability implementation mechanisms. In: Proceedings of International Workshop on Product Line Engineering (PLEES), pp. 59–64 (2002)
3. Capilla, R., Martínez, M.: Software architectures for designing virtual reality applications. In: 1st European Workshop on Software Architectures (EWSA). LNNC, vol. 3047, pp. 135–147. Springer (2004)
4. Czarnecki, C., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In: 3rd International Conference on Software Product Lines (SPLC). LNCS, vol. 3154, pp. 266–283. Springer (2004)
5. Gomaa, H., Shin, M.E.: Automated software product line engineering and product derivation. In: Proceedings of the 40th Hawaii International Conference on System Sciences (HICSS), p. 285 (2007)
6. Araújo Aleixo, F., Aranha Freire, M., Camara dos Santos, W., Kulesza, U.: Automating the variability management, customization, and deployment of software processes: a model driven approach. In: ICEIS 2010. LNBIP, vol. 73, pp. 372–387. Springer (2011)
7. Sinnema, M., Deelstra, S., Hoekstra, P.: The COVAMOF derivation process. In: International Conference on Software Reuse (ICSR). LNCS, vol. 4039, pp. 101–114. Springer (2006)

8. Cirilo, E., Nunes, I., García, A., de Lucena, C.J.P.: Configuration knowledge of software product lines: a comprehensive study. In: Proceedings of the 2nd International Workshop on Variability & Composition (VARICOMP), pp. 1–5. ACM DL (2011)
9. Thiel, S., Ferber, S., Fischer, T., Hein, A., Schlick, M.: A case study in applying a product line approach for car periphery supervision systems. In: Proceedings of In-Vehicle Software 2001 (SP-1587), pp. 43–55 (2001)
10. van Ommering, R., van der Linden, F., Kramer, J., Magee, J.: The Koala component model for consumer electronics software. IEEE Comput. **33**(3), 78–85 (2000)
11. Asikainen, T., Soininen, T., Männistö, T.: A Koala-based approach for modelling and deploying configurable software product families. In: Proceedings of the 5th International Workshop on Product Family Engineering (PFE-5). LNCS, vol. 3014, pp. 225–249. Springer (2003)
12. Asikainen, T., Männistö, T., Soininen, T.: Using a configurator for modelling and configuring software product lines based on feature models. In: Männistö, T., Bosch, J. (eds.) Proceedings of Software Variability Management for Product Derivation – Towards Tool Support, a Workshop in SPLC 2004, pp. 24–35. Helsinki University of Technology, Espoo, Finland (2004)
13. Raatkainen, M., Soinien, T., Männistö, T., Mattila, A.: Characterizing configurable software product families and their derivation. Softw. Process Improv. Pract. **10**(1), 41–60 (2005)
14. Cirilo, E., Nunes, I., Kulesza, U., Nunes, C., de Lucena, C.J.P.: Automatic product derivation of multi-agent systems product lines. In: Proceedings of the ACM Symposium on Applied Computing (SAC), pp. 731–732. ACM DL (2009)