Technical Report 2011-14 (V. 4)

SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems

Authored by:
Razieh Behjati, Tao Yue, Lionel Briand and Bran Selic

Date: September 16, 2012

Executive Summary

Context. Integrated control systems (ICSs) are heterogeneous systems where software and hardware components are integrated to control and monitor physical devices and processes. A family of ICSs share the same software code base, which is configured differently for each product to form a unique installation and, therefore, a large number of interdependent variability points are introduced by both hardware and software components. Due to the complexity of such systems and inadequate automation support, product configuration is typically error-prone and costly.
Objective. To overcome these challenges, we propose a UML-based product-line modeling methodology that provides a foundation for semi-automated product configuration in the specific context of ICSs.
Method. We performed a comprehensive domain analysis to identify characteristics of ICS families, and their configuration challenges. Based on this we formulated the characteristics of an adequate configuration solution, and derived from them a set of modeling requirements for a model-based solution to configuration. The SimPL methodology is proposed to fulfill these requirements.
Results. To evaluate the ability of SimPL to fulfill the modeling requirements, we applied it to a large-scale industrial case study. Our experience with the case study shows that SimPL is adequate to provide a model of the product family that meets the modeling requirements. Further evaluation is still required to assess the applicability and scalability of SimPL in practice. Doing this requires conducting field studies with human subjects and is left for future work.
Conclusion. We conclude that configuration in ICSs requires better automation support, and UML-based approaches to product family modeling can be tailored to provide the required foundation.

Contents

List of Figures

List of Tables

1. Introduction

Modern society is increasingly dependent on *integrated control systems* (ICS). These systems are large-scale, highly-hierarchical, heterogeneous systems-of-systems, where software and hardware are integrated to control and monitor physical devices and processes. Examples of such systems include oil and gas production platforms, industrial robots, and automotive systems.

To achieve higher quality and to reduce the overall engineering effort and production costs, many organizations in the ICS domain resort to various reuse strategies. In particular, many organizations have adopted software product-line engineering approaches [20, 30, 36, 27] to develop the software embedded in their systems. These product lines (i.e., product families) typically consist of a large variety of reusable hardware and software components that comprise a large number of interdependent *configurable parameters*. Product development, in this context, is done through *configuration*, which is the process of selecting and customizing the reusable components (through assigning values to their configurable parameters) according to the specific needs of a particular product.

Effectiveness of a product-line engineering approach is characterized by the quality of its support for *abstraction* and *automation* [26]. Abstraction, in general, plays a central role in software reuse. Concise and expressive abstractions are required to effectively specify collections of related reusable artifacts. Automation, on the other hand, is required for effective and reliable selection and customization of reusable components. As the complexity of systems increases, and the product lines grow (i.e., the numbers of reusable components and their configurable parameters increase), automation support becomes crucial to the configuration process. In practice, however, many cases of product-line engineering in the ICS domain lack concise and communicable abstractions of their reusable artifacts, and define architecture-level configuration processes that involve manually selecting and customizing components.

The complexity of integrated control systems and inadequate automation support result in increased likelihood of configuration errors. It is often difficult to ensure that the configuration data for a desired product is valid and internally consistent. Moreover, configuration errors are very costly and difficult to locate and fix, therefore making debugging processes expensive and lengthy.

A solution to the aforementioned configuration problems should both enable creating concise architecture-level abstractions of ICS families and provide automation support that ensures safe configuration of software in the ICS domain. In this paper, we focus on the formers and propose a modeling methodology, named *SimPL* (Simula Product Line), to create models of ICS families. The SimPL methodology serves as a first step to the development of a model-based and semi-automated configuration solution that we describe in this paper.

The SimPL methodology provides a notation and a set of guidelines for modeling commonalities and variabilities in ICS families. In particular, this methodology provides an architecture-level variability modeling approach that uses standard UML features for modeling configurable parameters, grouping them, and specifying their relationships. Relying on UML as a well-known industry-standard modeling notation for modeling both commonalities and variabilities allows extensive reuse of existing UML expertise, model analysis technologies (e.g., model validation and transformation), and tools. The design of SimPL was driven by a set of modeling requirements carefully identified from characteristics of ICS families, their configuration challenges, and characteristics of an adequate configuration solution in our context. The SimPL methodology was proposed because, according to our evaluation, none of the existing variability modeling approaches fulfill all of the identified modeling requirements.

The main contributions of this work are:

- A systematic analysis of the ICS domain to characterize ICS families and to identify and formulate the configuration challenges in ICS families.
- Definition of an adequate configuration solution in the ICS domain, based on our formulation of the problem.
- Derivation of a set of modeling requirements based on the characteristics of the adequate configuration solution. These requirements are intended to ensure that product-family models can provide the foundation required for developing automation support for configuration.
- Development of a UML-based modeling methodology, i.e., SimPL, that provides a notation and a set of guidelines to fulfill the above modeling requirements.
- An initial evaluation of SimPL by applying it to a large-scale industrial case study. To the best of our knowledge, only few applications of architecture-level product-line modeling approaches on industrial case studies have been reported in the literature.

Our evaluation of capabilities of the SimPL methodology indicates that the methodology satisfies all the modeling requirements for the subject of our case study, which is a representative ICS family. Furthermore, as reported in [8], our automated configuration approach, which is based on the SimPL methodology, can help address the configuration challenges in the ICS domain.

In the remainder of this technical report, we first describe an overview of our research approach in Section 2. An explanation of the industrial context together with our formulation of the problem is presented in Section 3. An adequate solution to the configuration challenges in the ICS domain is described in Section 4. The SimPL methodology is presented in detail in Section 5. Configuration in a model-based context is briefly explained in Section 6. We evaluate our modeling methodology in Section 7. Related work is presented and analyzed in Section 8. Finally, we conclude our work, and discuss directions for future work in Section 9.

2. Motivation and scope

The work presented in this paper is based on a collaboration with an industry partner, FMC technologies[1], and is therefore rooted in realistic contexts and problems. We strive to devise solutions for improving the software development process at FMC. This company delivers families of subsea oil production systems. Their software development process entails configuring a parameterized code base, a common practice in the ICS domain. Configuration, in this context, is complicated and challenging due to a number of factors, including the complexity of ICSs and inadequacies in the adoption of product-line engineering approaches. To devise a solution, we followed an engineering design process that is depicted in Figure 1.

[1]FMC Technologies, Inc. http://www.fmctechnologies.com/.

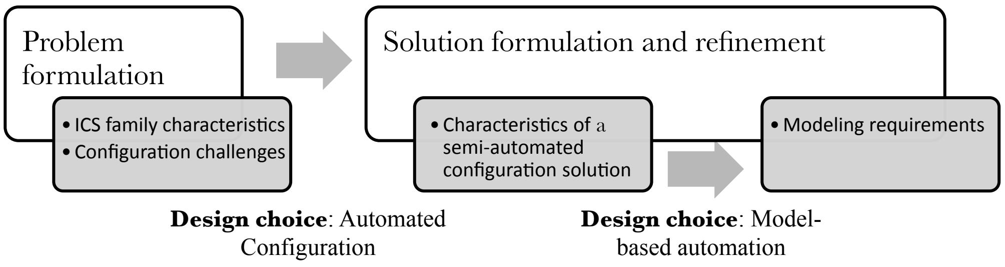

Problem formulation
• ICS family characteristics
• Configuration challenges
Solution formulation and refinement
• Characteristics of a semi-automated configuration solution
• Modeling requirements
Design choice: Automated Configuration
Design choice: Model-based automation

Figure 1: The engineering design process that we followed in our research.

As shown in Figure 1, we started the process by providing a clear formulation of the problem. To do so, we performed a systematic domain analysis of current practices of our industry partner, with the aim to identify the key characteristics of ICS families, and the associated software configuration challenges and their causes. Findings of this domain analysis, which are generalizable to many ICS families, are reported in Section 3.

We then pursued by characterizing an adequate solution to the configuration challenges that were identified and formulated during the problem formulation step. As shown in Figure 1, *automated configuration* and *model-based automation* are two major design choices that we made for addressing the configuration challenges. Automation is identified as one of the key requirements for product configuration and derivation [31]. In general, automation is a good solution to repetitive tasks that can be mechanized using a limited number of facts and relations. In the case of supporting software configuration for an ICS family, these facts and relations include configurable parameters, their interdependencies, and other information about the ICS family. To provide automation support, this information must be precisely and systematically collected, managed, and represented. For this purpose, we chose to use a model-based approach, since it provides a systematic, consistent, effective, and well-understood technique for capturing this type of information. To enable the required automation for configuration in the ICS domain, such a model-based approach should fulfill certain requirements. Characteristics of the adequate configuration solution, and the associated modeling requirements are presented in Section 4.

Figure 2 shows an overview of an adequate configuration solution, which incorporates the two design choices discussed above. This solution has a *product-family modeling step*, and a *semi-automated configuration step*. During the former, a generic model of an ICS family is built. Models created in this step should fulfill the modeling requirements mentioned above. Our assessment of the existing product-family modeling approaches, presented in Section 8, shows that none of these approaches can fulfill all the modeling requirements. Therefore, we have proposed a new modeling methodology, namely the SimPL methodology, to address the modeling requirements in our context. The SimPL methodology is explained in details in Section 5.

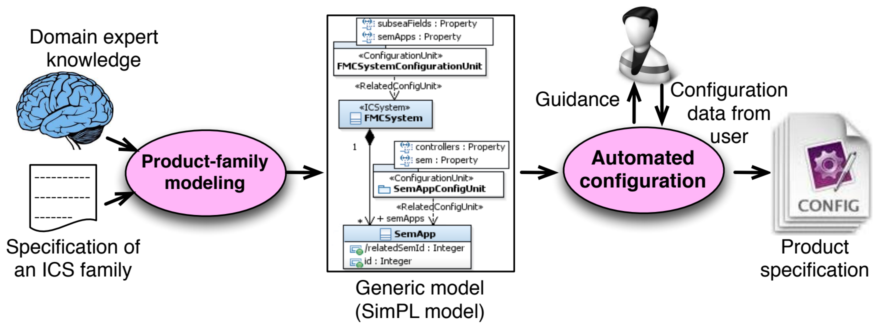


Figure 2: An overview of our model-based and semi-automated configuration solution.

The second step in Figure 1 (the semi-automated configuration step) entails a configuration engine that uses generic models of ICS families to provide automation support for software configuration. This paper focuses on the first step of the configuration solution in Figure 2 and only briefly describes the semi-automated configuration step. Details of the latter are presented in [8].

3. ICS families: characteristics and configuration challenges

In this section, we present the key characteristics of ICS families (Section 3.1), their typical configuration processes (Section 3.2), and the challenges of configuring software in large-scale ICS families (Section 3.3). These provide the context and rationale for the decisions that we made during the development of the SimPL methodology.

3.1. Characteristics of ICS families

Figure 3 shows a simplified model of a subsea "Christmas" (Xmas) tree. A subsea Xmas tree in a subsea oil production system provides mechanical, electrical, and software components for controlling and monitoring a subsea well. In particular, a subsea Xmas tree (e.g., xt in Figure 3) contains a subsea control module (e.g., scm), and a number of mechanical and electrical devices (e.g., s1, s2, and v1). A subsea control module contains subsea electronic modules (e.g., semA and semB), and software applications deployed on them (e.g., semAppA and semAppB). Mechanical and electrical devices in the Xmas tree are controlled and monitored by these software applications. Therefore, software applications deployed on subsea electronic modules are configured, mainly, based on the number, type, and other details of the related devices (e.g., sensors and valves).

To identify the characteristics of families of ICSs, we studied three different types of subsea oil production systems belonging to the same product family developed by FMC. These three systems, carefully selected with the help of FMC engineers, are representative in the sense that they reuse and configure most of the configurable components of the generic product. We had many face-to-face discussions with FMC engineers, and studied all relevant technical documents, defect tracking systems, hardware design schematics and source code of the software components. Based on the results of this domain analysis, we identified a set of characteristics of the family of subsea oil production systems, which can be generalized to cover many other types of ICSs:

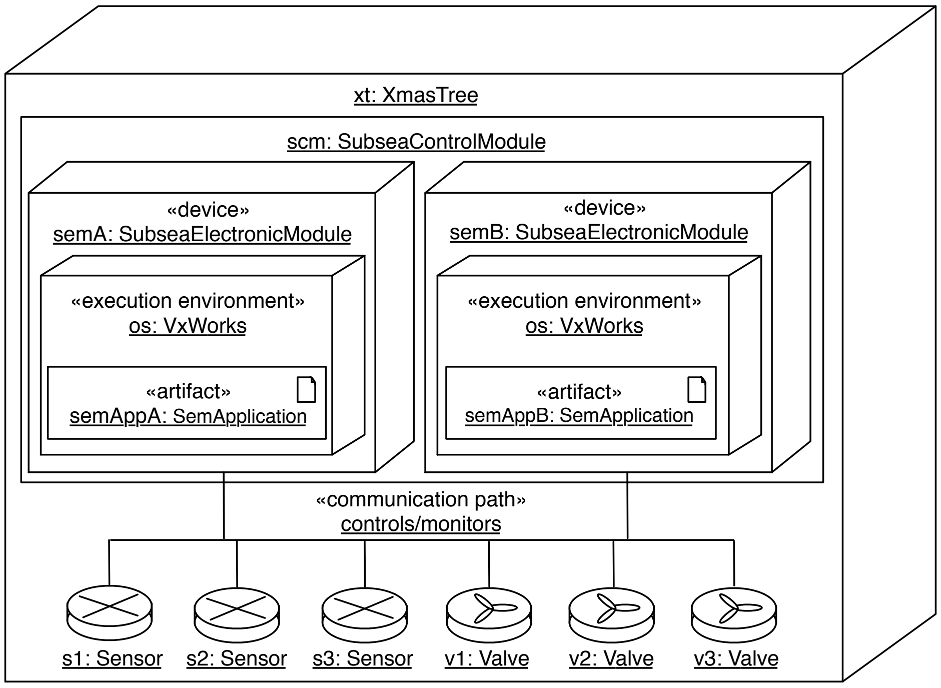


Figure 3: A simplified model of a subsea control module.

IC1. **Heterogeneous, large-scale, and hierarchical systems.** ICSs are heterogeneous systems that typically combine mechanical, electrical, and software components. ICSs are large-scale both with respect to the diversity of the types of their contained hardware and software components (i.e., dozens of component types) and the number of components that a system typically contains (i.e., thousands of hardware and software components). ICSs are hierarchical, with complex components containing other finer-grained components, and so on.

IC2. **Configurable hardware.** In an ICS family, the hardware topology can vary from one product instance to another, with each topology being a specific configuration of the generic hardware design of the family. Mechanical and electrical engineers design the hardware topology (i.e., configure the hardware) based on customer requirements, environmental conditions, and different regulations and standards. During hardware configuration, (sub)components at various (and possibly all) levels of the hardware hierarchy are configured.

IC3. **Configurable software code base.** In an ICS product, the software application is responsible for controlling and monitoring hardware devices. Usually, ICS products belonging to a family share a generic software code base (e.g., a set of highly-parameterized C++ classes). This generic code base is instantiated and initialized differently for each product, mainly based on the hardware configuration. For example, the number of electrical and mechanical devices in the hardware configuration of a product, as well as their properties (e.g., specific sensor resolution and scale levels) affect the number and values of run-time objects in the software configured for that product. Software configuration in ICS families is, therefore, the process of building *configuration files* (e.g., makefiles or boot files) that contain the information required for initializing and running a unique installation (i.e., a set of deployable and executable software modules) of the code base for a specific product. Software configuration engineers (the individuals who specify the configuration of software), assign values to tens of thousands of configurable parameters at various levels of the software hierarchy, based on the hardware configuration and their domain knowledge, to create the configuration files.

IC4. **Interdependability of the configurable parameters.** Many dependencies exist among configurable parameters, especially between the ones introduced by the software and the ones introduced by the hardware. Similar to the configurable parameters, these dependencies exist at various levels of the software or hardware hierarchies.

3.2. Configuration process in practice

Before explaining configuration challenges in the ICS domain, we first briefly describe the main aspects of the product configuration process in practice. Product configuration for ICS families includes: (1) configuring the hardware by making decisions about the number, type, and other properties of electrical and mechanical components (e.g., computing resources, and devices), as well as designing the hardware topology, and (2) configuring the software that controls and monitors hardware devices and processes.

Separated hardware and software configuration processes. Hardware and software configurations are performed separately, after the tendering and front-end engineering phases [37]. First, electrical and mechanical engineers configure the hardware and produce schematics of the customized hardware design. Then, software configuration engineers configure the software according to the hardware schematics, their own domain knowledge, and the product-specific software requirements derived during the tendering and front-end engineering phases. Software configuration is largely governed by a set of *consistency rules* that specify constraints on software and hardware components and dependencies among the components and their configurable parameters.

Manual configuration. Due to historical, organizational, and technical reasons, software configuration in the ICS domain is largely manual. In particular, software configuration engineers have to manually ensure the consistency of the values assigned to tens of thousands of interdependent configurable parameters.

Multiple configuration files Usually, more than a single configuration file is created for each product during its development lifecycle. For example, for each phase of testing, a different configuration file is created by instantiating and configuring only the components that are involved in that particular testing phase.

3.3. Challenges in the software configuration process

Our analysis of the defect tracking systems at our industry partner shows that software configuration is an error-prone and costly process. Similar observations have been reported in the literature [17]. This is largely due to the large number of configurable parameters, their interdependencies, and the many stakeholders involved in the manual software configuration process. Our analysis led to the identification of the following challenges regarding the configuration of ICS families:

CC1. **Tacit knowledge and inadequate documentation.** Configuring the software application so that it matches the hardware configuration of a product requires accurate knowledge about the generic software, the hardware design, its particular configuration for the product, the dependencies between hardware and software components, and other consistency rules. In companies that have gradually adopted a product-line engineering approach, a systematic, complete, and up-to-date documentation about these is often inadequate or missing altogether. This results in a lack of shared understanding of the system among different stakeholders (e.g., software developers, hardware designers, and configuration engineers) involved in the product design and configuration. In particular, it is not practical to expect configuration engineers to have complete knowledge of the system design, all the design constraints, and all the consistency rules. The required configuration-related knowledge is scattered hidden in the minds of various specialty engineers or across documents.

CC2. **Insufficient configuration guidance.** At our industry partner, configuration engineers are provided with a set of guidelines to help them with software configuration. However, these guidelines are often incomplete, outdated, complex, and sometimes complicated and hard to follow. Moreover, it is usually unrealistic to expect – and the complexity of the configuration guidelines makes it even less probable – that every configuration engineer strictly follows all the guidelines.

CC3. **Lack of automated verification.** The scale of ICSs (i.e., thousands of components and tens of thousands of configurable parameters) and their complexity (i.e., heterogeneous systems with large numbers of dependencies between hardware and software) make the configuration process a difficult and overwhelming task for configuration engineers. This, together with a lack of interactive support for automated verification of the configuration data, results in many chances for human errors.

CC4. **Insufficient support for configuration reuse.** Certain subsystem or components of an ICS may have identical or similar configurations, for example, due to the redundancy required for fault tolerance. Consequently, identical or nearly identical configuration patterns have to be entered repeatedly. Lack of automation and inadequate support for reuse (e.g., only through a copy-and-paste mechanism) can result in inconsistencies, as well as costly rework during the software configuration.

CC5. **Expensive debugging of configuration data.** The lack of an interactive support for automatically verifying the configuration data leads to configuration errors that are discovered very late, either when the configuration is completed, or later during testing. In many cases, due to the large-scale and complex nature of ICSs, configuration errors are mistakenly reported as software errors or integration errors (e.g., interface mismatch between hardware and software), making it difficult and costly to locate and fix the problem.

CC6. **Improper synchronization mechanism.** As mentioned in Section 3.2, to meet the needs of different steps of testing and production, several configuration files are created for each product. These configuration files should be kept synchronized and consistent throughout the production lifecycle. Currently, at our industry partner, these configuration files are treated as distinct assets, without any well-defined mechanism to keep them synchronized. This may have undesirable consequences. For example, configuration bugs that are fixed in one configuration file are not guaranteed to be fixed in others, resulting in inconsistencies, and repetition of errors.

CC7. **Evolution of the product family and outdated configurations.** Our experience with our industry partner, consistent with what is reported in the literature [18, 11, 17], shows that industrial ICS families are constantly evolving. Evolution of the product family may, for example, contain a change in the software code base, which can lead to inconsistent and outdated configurations.

In the work presented in this paper, we have narrowed our scope to finding a solution to the first five challenges of the software configuration process in ICSs. This is not because the remaining two issues are deemed unimportant, but simply due to limitations on available time and resources. For simplicity, in the remainder of this technical report, we use the term *configuration challenges* to refer to the first five challenges mentioned above.

4. Overview of a model-based, semi-automated configuration solution

A complete solution to all the configuration problems at our industry partner entails improvements in both the methodological aspects of the product configuration process (e.g., separation of software and hardware configuration subprocesses, and multiple configuration files for each product) and the automation support provided for the product configuration process. Our research focuses only on the latter. To this end, we opted for a semi-automated configuration solution (see Figure 2) designed to address the configuration challenges described in Section 3.3. This solution is based on concise architecture-level abstractions of product families in the ICS domain. Our work to date can, therefore, be considered as a first significant step towards a complete solution.

In the following, we first describe, in Section 4.1, the characteristics of the adequate configuration solution mentioned in Section 2. We present the main configuration-related modeling concepts in Section 4.2. Modeling requirements for enabling the automation support are formulated in Section 4.3. Finally, in Section 4.4, we describe several practicality requirements that a modeling solution must meet in order to be applicable in practice. Together, these requirements are used in the remainder of the technical report as a basis for justifying the SimPL methodology and evaluating the suitability of existing product-line modeling approaches.

4.1. Characteristics of an adequate configuration solution

The configuration solution in Figure 2 is, mostly, aimed at reducing the likelihood of human errors during configuration, by interactively guiding configuration engineers throughout the configuration process, automatically verifying configuration decisions, and, automating, to a certain extent, decision making and configuration reuse. The characteristics of such a configuration solution are listed below. These characteristics are derived from the general ICS characteristics and the configuration challenges described in Section 3. Consequently, they are not specific to our solution, but may be more broadly applicable.

Ch1. **Automation.**

(a) *Automated stepwise verification of configuration decisions.* To reduce chances of faulty configurations and to enable early detection of configuration errors, a configuration solution should enable automated and iterative verification of configuration decisions. Such verification support proactively guarantees the correctness and consistency of configuration decisions with respect to the interdependencies of configurable parameters and other consistency rules governing the product family.

(b) *Interactive guidance throughout the configuration process.* Configuration guidelines should be implemented and enforced by the configuration solution. In particular, the configuration solution must (1) suggest consistent values to be assigned to configurable parameters that, in general, take their values from finite domains, and (2) guide the user throughout the software configuration process by directing the order of variability resolution steps, such that the effort required for making configuration decisions and the cost of consistency checking are minimized.

(c) *Automated decision making.* New configuration decisions can be inferred from previously made decisions and the interdependencies among the configurable parameters. A configuration solution should be able to detect such situations and automatically infer new configuration data that is consistent with previously made decisions. This can reduce the manual configuration effort and improve the quality of configuration data by reducing inconsistencies.

In special cases, where interdependencies imply identical values for two or more configurable parameters, the configuration decision made for one of the configurable parameters can be reused for the others. We refer to this as the reuse of configuration data. This is particularly important when hundreds or even thousands of such configuration parameter values can be reused.

Note that the term automation in the remainder of the technical report only refers to the three characteristics described above and does not imply a fully automated software configuration approach.

Ch2. **Completeness.** The output of the configuration process is a configuration file that specifies how the generic code base must be instantiated and initialized for a particular product. A configuration solution must enable collecting and representing all the information – configuration decisions about all the configurable parameters – required for generating such an output.

Ch3. **Scalability.** The large-scale nature of ICS families and their product instances impacts efficiency and applicability of a semi-automated configuration solution in practice. We consider two aspects of scalability in the design of our configuration solution.

(a) *Large-scale product families*. A configuration solution must be able to handle the large diversity in the types of components and configurable parameters contained by an ICS family.

(b) *Large-scale products.* A configuration solution must be able to efficiently deal with (e.g., through automation) the consistent configuration of the large number of interdependent configurable parameters.

4.2. Modeling to support semi-automated configuration

Providing the automation described in the previous section requires precise knowledge about the product family. As noted earlier, we use a model-based approach to provide this knowledge. A model is an abstraction of a system, which retains only information that is relevant for a specific purpose. Using models, therefore, allows working with relatively simple specifications of a system, instead of dealing with the complexities of the actual system. Note that an abstraction of a system can still contain precise information required for a particular purpose, which in this case is providing a semi-automated configuration solution characterized in Section 4.1.

Figure 4 is a conceptual model, in the form of a UML class diagram, that shows the main concepts involved in the model-based configuration of product families, as well as the relationships among those concepts. The conceptual model is included here to help understand modeling requirements in Section 4.3, and to clarify the terminology used in Section 5, where we explain SimPL.

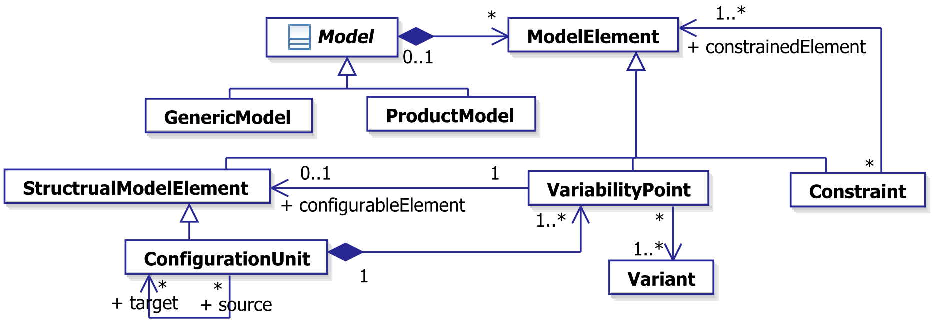


Figure 4: The conceptual model of a model-based configuration solution.

A *GenericModel* is part of the description of a product family, and captures all commonalities and variabilities of the subject product family. A *ProductModel*, on the other hand, captures the specification of a specific product instance. Both *ProductModel* and *GenericModel* are subtypes of *Model*.

A *Model* is composed of a number of *Model element*s. *StructuralModelElement*, *VariabilityPoint*, and *Constraint* are three subtypes of *ModelElement* that are required for modeling variabilities.

A *StructuralModelElement* specifies a structural aspect or element of a system. Structural elements include components, subcomponents, and their properties. A *StructuralModelElement* can represent a configurable element of the system. In the specification of a product family, a configurable element is a model element the value of which can vary from one product to another.

A *VariabilityPoint* refers to a *StructuralModelElement* representing a configurable element (i.e., VariabilityPoint::configurableElement). A variability point is a place in the specification of a product family (i.e., a generic model) where a specific decision has been narrowed down to multiple options, but the option to be chosen for a particular system has been left open [6][2]. Variability points, in our context, can be instantiated[3]. A configurable parameter represents a particular instance of a variability point. Only *GenericModel*s can own *VariabilityPoint*s. *ProductModel*s do not have any *VariabilityPoint*s. However, this constraint is not explicitly captured in Figure 4.

A *Constraint* is a model element that refines the semantics of another model element or defines dependencies between two or more model elements.

For each *VariabilityPoint*, there is a set of possible *Variant*s. A variant is one valid option for a variability point. The set of valid variants can be specified in several ways, including value ranges, constraints, or enumerating literals. The set of valid variants for a variability point depend on the type and other details of the respective configurable element. When resolving an instance of a *VariabilityPoint*, a *Variant* is bound to the respective configurable element.

4.3. Modeling requirements

The goal in the product-family modeling step in Figure 2 is to provide domain experts with a suitable notation and guidelines for creating product-family models that can enable semi-automated configuration. In order to define precise objectives for product-family modeling, we discuss below the requirements that such a modeling solution should fulfill.

Req1. **Comprehensive variability modeling.** Configuration is all about assigning values to configurable parameters. To ensure completeness of the approach and to provide automated verification, guidance, and value inference throughout the entire configuration process, the modeling methodology must enable capturing all types of variabilities, and all types of interdependencies among them.

Req2. **Software modeling.** The ultimate goal of software configuration is to instantiate and initialize the generic code base into a particular software product. The generic model of the product-family should contain a software model capturing an abstraction of the generic code base and its configurable elements. The modeling methodology must, therefore, provide notation and guidelines for capturing and properly locating all the configurable elements in the software model. To cope with the highly-hierarchical nature of ICS families, the modeling notation must enable the hierarchical organization of software modules and classes.

Req3. **Hardware modeling.** In the context of ICS families, many decisions about the software configuration are a direct consequence of the underlying hardware configuration. Modeling the hardware and its variability points can, therefore, enables reusing some of the hardware configuration decisions to

[2]In the literature, the term variation point is usually used instead of variability point.

[3]This is because, in our context, generic models are class-based models. To do the configuration, everything in the generic models, including variability points, should be instantiated first. Instances are the elements that can be configured.

automatically create an initial version of the software configuration reflecting the main aspects of the hardware configuration. The modeling methodology must, therefore, address mechanical and electrical components. In particular, the hardware modeling notation should be expressive enough to capture electrical components on which configurable software is deployed and those devices that are controlled by, or, more generally, interact with software. Similar to software modeling, the modeling notation for hardware must enable the hierarchical organization of hardware components.

Req4. **Modeling software-hardware dependencies.** To enable the consistent reuse of hardware configuration decisions for creating the initial software configuration, the modeling methodology must be able to precisely capture the dependencies between hardware and software in an ICS family. These dependencies include software to hardware deployment and software-hardware interactions for the purpose of controlling and monitoring devices and instruments.

Req5. **Traceability of variability points to software and hardware model elements.** Variability points should be mapped to software and hardware model elements where the variability takes place. This enables modelers to reuse the information captured in software and hardware models to capture the relationships between variability points. This is possible because most of the dependencies among variability points are due to the specific variability that they specify in the system, which is modeled in software and hardware models. This traceability is required to enable automatic checking of consistency of configuration decisions with respect to the dependencies and constraints defined in the model. In addition, for variability points that represent parameters of the software code base, such traceability is required to enable instantiation and initialization of the code base from configuration data. The modeling methodology must, therefore, effectively enable the traceability of variability points to their corresponding configurable elements in the software and hardware models.

Req6. **Hierarchical organization and grouping of variability points.** To handle configuration of large-scale products that involves assigning values to tens of thousands of configurable parameters, the variability points must be hierarchically organized and grouped. To help a configuration engineer better relate the hierarchy of variability points to the domain, the hierarchy of variability points must reflect the software and hardware hierarchies. In other words, it is better to organize and group the hierarchy of variability points in the way that it is similar to the hierarchy of their corresponding configurable software and hardware elements. The modeling methodology must, therefore, provide a modeling notation to capture such hierarchy of variability points.

4.4. Practicality requirements

For our approach to be applicable in practice, the modeling solution should, as well, fulfill the following practicality requirements:

PR1. **Simplicity.** The modeling notation, in addition to being expressive enough, should be simple and easy to learn and apply. This would help reduce the training cost, and therefore increase the applicability of the methodology.

PR2. **Tool support.** An important practical consideration is the ready availability of tool support. In general, modeling, and, in particular, modeling large-scale systems is laborious. Effective tool support can therefore help ease the modeling step.

PR3. **Extensibility.** In practice, it should be possible to augment the generic model devised for an ICS family. To do so may require to extend our notation to support additional requirements, such as facilitating forward engineering activities (e.g., testing) or automated code generation.

5. The SimPL modeling methodology

In this section, we present the SimPL methodology, which provides notation and guidelines that are particularly designed to fulfill the modeling requirements and the practicality requirements presented in Sections 4.3 and 4.4. First, we provide an overview of the SimPL methodology in Section 5.1. The modeling notation of SimPL and a brief explanation of the overall process of applying it are presented in Sections 5.2 and 5.3. Additional explanation of the process and the use of the modeling notation is given in Sections 5.4 through 5.6.

5.1. Overview of the SimPL methodology

An overview of the SimPL methodology explaining how it addresses the practicality and the modeling requirements is provided in this section.

A standard-based methodology to fulfill practicality requirements

The modeling notation in the SimPL methodology is simple. It consists of (1) a subset of structural model elements of UML 2 [3], (2) four stereotypes from the standard MARTE profile [1], and (3) 10 additional stereotypes defined in a lightweight UML profile, named SimPL. UML, the base modeling notation in the SimPL methodology, is a widely-accepted and widely-taught industry-standard modeling notation. The four stereotypes from MARTE provide standard facilities for modeling hardware. The stereotypes defined in the SimPL profile extend UML packages and dependencies and support separation of concerns and variability modeling.

UML is supported by a wide range of open-source and commercial tools. Many of these tools can efficiently handle large-scale models with tens of thousands of model elements. Relying on UML, therefore, allows reuse of existing tools, either commercial or open source, and ensures the scalability of our approach from a technical/technological standpoint. Moreover, relying on a well-known standard allows us to benefit from many technologies around it, for example for checking the consistency of models (e.g., [19], also see [35]).

UML is a general-purpose modeling language. It provides a wide range of constructs, and a generic extension mechanism that allows tailoring the UML metamodel for a specific domain. These two factors together provide the necessary foundation for making our modeling methodology extensible.

A multi-view, UML-based methodology to fulfill modeling requirements

The SimPL methodology provides a multi-view and UML-based modeling approach to enable modeling large-scale ICS families, while fulfilling the modeling requirements presented in Section 4.3.

In the SimPL methodology, we organize the generic model of an ICS family into multiple views. This enables us to offer separation of concerns (by presenting to different stakeholders only the portion of the family model that is relevant to them), while at the same time ensuring the consistency of the whole family model as a unified artifact.

Figure 5 is an excerpt of the SimPL metamodel[4] depicting different views that are used to organize the generic model of an ICS family. As shown in Figure 5, the generic model of an ICS family is partitioned

[4]The complete metamodel of the SimPL methodology can be found in Appendix A.

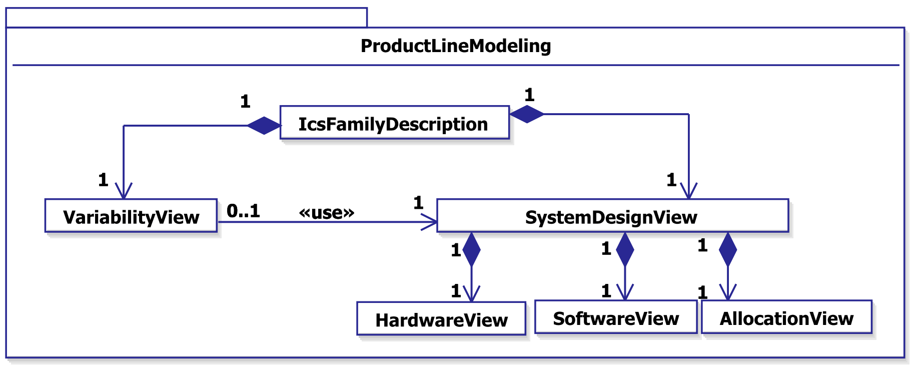


Figure 5: Multiple views introduced by the SimPL methodology.

into two main views: the *System design view*, and the *Variability view*. To be consistent with the terminology commonly used in the literature, we use *base model* to refer to the portion of the generic model that is covered by the system design view, and *variability model* to refer to the portion of the generic model that is covered by the variability view. The «use» dependency between *VariabilityView* and *SystemDesignView* addresses the requirement that variability points in the variability model must be traced back to the elements (i.e., configurable elements) in the base model.

As shown in Figure 5, the base model is further split into three sub-views: the *Software view*, the *Hardware view*, and the *Allocation view*. In the remainder of this technical report, *software model* refers to the set of model elements covered by the software view. Similarly, the terms *hardware model* and *allocation model* refer to the model elements covered by the hardware view and the allocation view, respectively.

The base model in the SimPL methodology is created using UML and MARTE. UML, as a general-purpose software modeling language, provides all the necessary constructs and abstractions required for modeling software. In addition, UML's extension mechanism (e.g., profiling) enables introducing new concepts and semantics to the language. MARTE is the UML extension that provides the concepts required for modeling hardware and the dependencies between hardware and software.

We use a refinement of UML, i.e., the SimPL profile, to create the variability model. For this purpose, the SimPL profile refines the UML template and package concepts. UML templates can be used to specify generic structures, and provide the necessary foundation for modeling variability points, as well as tracing them back to software and hardware model elements. Packages in UML are used to group and organize model elements, and provide the necessary foundation for grouping and hierarchically organizing the variability points.

To completely fulfill the modeling requirements, we provided a set of modeling guidelines that should be followed when creating the base and variability models. The SimPL profile provides context-specific stereotypes, attributes and constraints to allow modelers to more easily follow the modeling guidelines provided by the SimPL methodology.

In summary, we rely on industry-standard modeling notations, i.e., UML and MARTE, to fulfill, to a large extent (if not completely), all the practicality requirements presented in Section 4.4. Simultaneously, these notations provide the necessary constructs for modeling software, hardware, and their dependencies. The SimPL profile together with inherent features of UML (i.e., templates and packages) enables comprehensive modeling of variability points, tracing variability points to software and hardware model elements, and grouping and hierarchically organizing the variability points. To fully meet the modeling requirements, we have provided a set of modeling guidelines, and implemented the SimPL profile as an aid to help modelers follow these guidelines.

In subsequent subsections, we describe the SimPL profile and the process of applying the SimPL methodology. Then, we describe each of the views introduced above. Associated with each view is a viewpoint specification. The full specification of the viewpoints can be found in Appendix C.

5.2. Modeling notation and the SimPL profile

As noted earlier, the modeling notation used in the SimPL methodology is based on UML and two extensions of it, MARTE and SimPL. The UML constructs that are necessary for creating the base model of an ICS family are classes, properties, and relationships (i.e., the generalization relationship, and several types of association relationships). We rely on UML templates and packages to create variability models. Moreover, UML packages are used to organize the generic model of an ICS family into views, sub-views, and design hierarchies in the sub-views. In the following, we first briefly introduce the MARTE stereotypes that are used in the SimPL methodology. Then, we describe the SimPL profile, which is particularly designed and implemented to fulfill the modeling and practicality requirements.

MARTE
Four stereotypes from MARTE are used in SimPL to create hardware models and to model software to hardware bindings/allocations. To distinguish between hardware and software classes, each class in the hardware model must be stereotyped by one of the MARTE stereotypes «HwComputingResource», «HwComponent», or «HwDevice». The MARTE stereotype «Assign» is used to model a software to hardware dependency (i.e., deployment, allocation, or binding). These stereotypes, along with their usage in the SimPL methodology, are described in Sections 5.4.2 and 5.4.3.

SimPL
The SimPL profile extends the UML metamodel to implement the SimPL metamodel presented in [10]. Moreover, to support hardware modeling, the SimPL profile imports the four MARTE stereotypes mentioned above. The SimPL profile defines:

1. Six stereotypes to enforce the multi-view organization of the generic model of an ICS family according to the metamodel presented in Figure 5.
2. A stereotype to identify the topmost element in the design hierarchy of the system.
3. Three stereotypes to refine UML template packages for variability modeling.
4. A set of consistency rules in the form of OCL constraints on the newly defined stereotypes to assist modelers follow the methodology and ensure, to some extent, the consistency of the resulting model[5].

[5]For this purpose, the modeling tool must allow validating profiled models and providing feedback on their validation. Some tools, e.g., IBM RSA 8 [4], provide this functionally, and are used in practice to enforce consistent use of a profile [29].

Six stereotypes to enforce multi-view organization
Figure 6 shows the six stereotypes used to organize the generic model of an ICS family. As shown in this Figure, five stereotypes are defined to represent the five views and sub-views of SimPL. All these stereotypes are subtypes of an abstract stereotype, named *View*. We refer to the stereotype «View» and its subtypes as *Viewpoint stereotype*s. To reuse UML's inherent mechanism for model organization, the viewpoint stereotypes extend the UML package construct. We refer to a package that is stereotyped by a viewpoint stereotype as a *Viewpoint package*.

Associated with each viewpoint stereotype is a set of OCL constraints. These OCL constraints represent the consistency rules mentioned above, and are used to refine the semantics of the viewpoint packages according to the needs of SimPL. Constraints associated with viewpoint stereotypes along with the specifications of each viewpoint are presented in Appendix B and Appendix C.

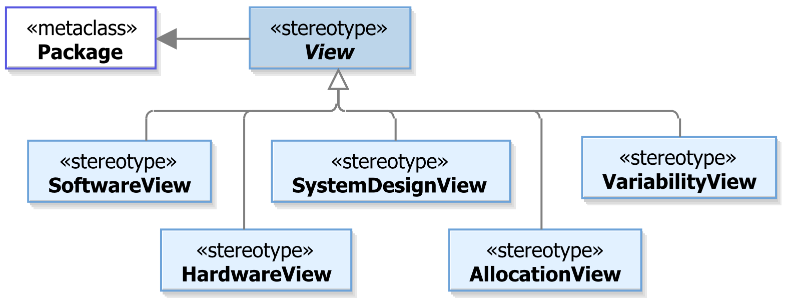


Figure 6: Viewpoint stereotypes (constrained stereotypes used to implement the SimPL methodology as a multi-view methodology).

A stereotype to indicate topmost element
Configuration of highly-hierarchical ICS families requires traversing the hierarchal structure in the generic this, we need a way to inform the configuration tool about the starting point of the configuration process. For this purpose, we have introduced the stereotype «ICSystem» (Figure 7), which indicates the topmost element in the base model. This stereotype must be applied to one and only one UML class in the base model of the ICS family. Note that this is not a constraint in any way, since even if a system may have multiple "top" elements, it is always possible to introduce an abstract top element on top of those and stereotype it as «ICSystem». Having exactly one class stereotyped by «ICSystem» in the base model is, however, a consistency rule that must be satisfied. To achieve this, we have modeled this consistency rule as an OCL constraint in the SimPL profile.

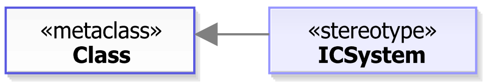


Figure 7: The «ICSystem» stereotype for denoting the topmost element.

Three stereotypes to assist variability modeling
The SimPL profile contains another set of stereotypes that are introduced to support variability modeling. Figure 8 shows these stereotypes and their relationships to UML metaclasses.

«ConfigurationUnit» is a stereotype that applies to UML packages. In the SimPL methodology, *configuration unit*s are UML template packages stereotyped by «ConfigurationUnit» that form the main build-

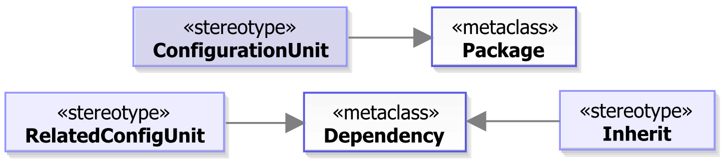


Figure 8: Stereotypes supporting variability modeling.

ing blocks for grouping and organizing variability points in the variability model. To reflect software and hardware hierarchies of the base model in the organization of variability points, each configuration unit in the variability model is associated with exactly one class in the base model. The class associated with a configuration unit is referred to as the *origin class* of that configuration unit. The relationship between a configuration unit and its origin class is realized in the SimPL profile using a dependency stereotyped by «RelatedConfigUnit». This stereotype can be applied only to those dependencies that connect a template package stereotyped by «ConfigurationUnit» to a class. An OCL constraint on «RelatedConfigUnit» is defined to ensure a meaningful application of this stereotype.

A configuration unit can inherit variability points from another configuration unit. This is enabled through a dependency connecting a sub-configuration-unit to its super-configuration-unit. This dependency should be stereotyped by the «Inherit» stereotype. This stereotype can be applied only to those dependencies that connect two UML packages both stereotyped by «ConfigurationUnit». An «Inherit» dependency between two configuration units implies that configuring instances of the origin class of the sub-configuration-unit, in addition to resolving variability points listed in the sub-configuration-unit, requires resolving variability points listed in the super-configuration-unit[6]. More details on UML template packages, the three stereotypes defined for variability modeling, and their usage in the SimPL methodology are provided in Section 5.6.

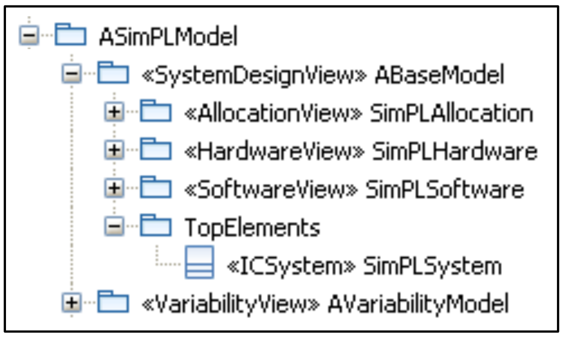
5.3. The overall modeling process

The following is the list of modeling activities that should be performed to create a *SimPL model*. A SimPL model is a generic model of an ICS family that is created by following the SimPL methodology, and taking advantage of the SimPL profile. Two excerpts of a SimPL model are provided in Figures 9 and 10 to illustrate the steps below. This SimPL model is elaborated in Section 5.4 in Figures 11 through 15.

MA1. The process of creating a SimPL model starts by creating a package structure, similar to the one in Figure 9. This package structure organizes the SimPL model according to the different views proposed by the SimPL methodology. Each package in this structure must be stereotyped by one viewpoint stereotype. Additional subpackages may exist in each of these packages to provide a finer-grained organization of the model.

MA2. In the second step, a class stereotyped by «ICSystem» must be created in the base model to represent the topmost element. This class represents a *configurable class* (i.e., one type of configurable element) and is composed of at least one software component and at least one hardware component. In Figure 10, FMCSystem represents the topmost element. SubseaField, and SemApp are two of the subcomponents of FMCSystem.

[6]Note that the configuration tool is responsible for ensuring that all the required variability points are resolved for an instance of an origin class. The family model is only used to provide the tool with the required information to do this.



ASimPLModel
«SystemDesignView» ABaseModel
«AllocationView» SimPLAllocation
«HardwareView» SimPLHardware
«SoftwareView» SimPLSoftware
TopElements
«ICSystem» SimPLSystem
«VariabilityView» AVariabilityModel

Figure 9: Sample package structure of a SimPL model.

MA3. Variability points of the topmost element must be captured in the variability model. Therefore, the modeler must create a configuration unit in the variability view and associate it with the topmost element in the base model. Modeling such a configuration unit in the variability view requires the *configurable properties* (i.e., one type of configurable element) of the topmost class to be captured as part of the base model. Modeling this information in the base model before creating the configuration unit is necessary because variability points in the configuration unit must be traced back to these configurable properties. Package FMCSystemConfigurationUnit in Figure 10 shows the configuration unit associated with the topmost element of the system, FMCSystem. Note that the template parameters of FMCSystemConfigurationUnit refer to the two properties of FMCSystem: subseaFields, and semApps, which must be modeled in the base model beforehand.

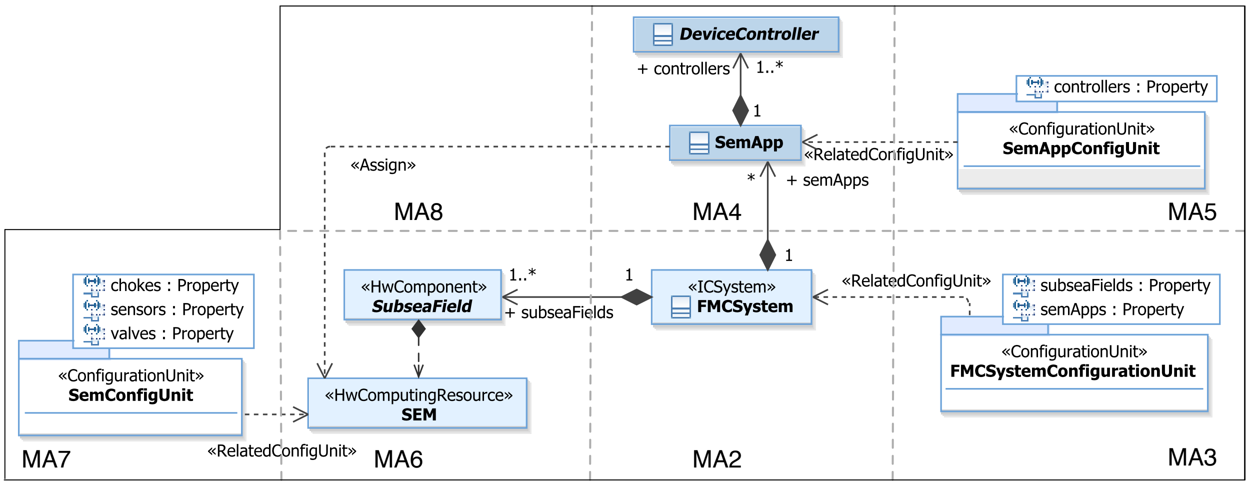


Figure 10: A summary of the main modeling activities.

MA4. Direct software subcomponents (e.g., SemApp) of the topmost element of a system are the classes at the roots of decomposition hierarchies in the software model. To create the software model, a modeler must start from such topmost software classes and decompose them into their subcomponents (e.g., DeviceController) and, model the relationships between them. Such decomposition hierarchies, along with taxonomies of software classes should be detailed enough to provide sufficient information for guiding configuration. In particular, all the configurable software classes must be reachable from the topmost element of the system through decomposition and taxonomic hierarchies in the software model.

MA5. For each configurable software class in the software model, a configuration unit (e.g., SemAppConfigUnit) must be created in the variability model and associated with the configurable software class, after all the necessary information (e.g., configurable properties of the configurable software class) is included in the software model.

MA6. Direct hardware subcomponents (e.g., SubseaField) of the topmost element of a system are the hardware elements serving as roots of decomposition hierarchies in the hardware model. Such decomposition hierarchies, along with taxonomies of hardware components and devices should be detailed enough to enable modeling software to hardware deployment and, to provide sufficient information for guiding configuration. For example, the SEM component must be captured in the hardware model, because it is the hardware computing resource to which SemApp is deployed. Note that, as shown in Figure 10, SEM is indirectly contained by the SubseaField.

MA7. Associated with each configurable hardware component in the hardware model, a configuration unit (e.g., SemConfigUnit) must be created in the variability model, after all the necessary information (e.g., configurable properties of the configurable hardware class) is added to the hardware model.

MA8. Allocation models in the SimPL methodology specify constraints on software to hardware deployment. To model the allocation relationship between a software class and a hardware class the two classes should be defined beforehand. The allocation of SemApp to SEM is shown in Figure 10, using a dashed line connecting the two classes[7]. This dashed line is the MARTE notation for an assignment, and is stereotyped by «Assign».

MA9. We use OCL constraints to model additional information that cannot be captured using classes and relationships. OCL constraints can be added to the base model at any point during the modeling process.

Note that the numbering in the list above does not imply a strict ordering of activities. For example, software and hardware models can be created in parallel. Similarly, as implied in MA3, MA5, and MA7, it is not required to create a complete base model before beginning variability modeling.

5.4. System design view

The base model that is presented via the system design view is a compilation of software, hardware, and allocation models. The base model contains the topmost element of the system and the decomposition of this element into its subcomponents. UML composition associations are used to model this decomposition. Subcomponents of the topmost element are either software components or hardware components, which are organized into software and hardware sub-views, respectively.

Figure 11 shows an example. FMCSystem is the topmost element. Software and hardware models are accessible through packages FMCHardware and FMCSoftware, respectively. Software subcomponents of FMCSystem are McsApp, and SemApp, and its hardware subcomponents are SubseaField, and MCS.

[7]This type of relationship does not always have to be entered graphically (since such a representation may not scale very well), but can also be entered by other means (e.g., via a special front-end tools).

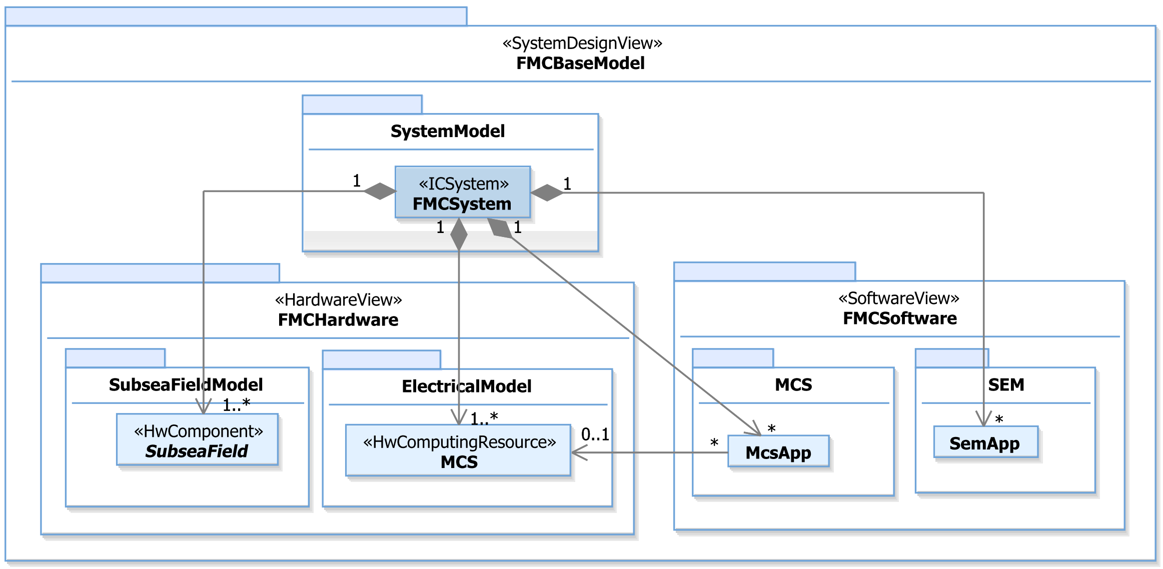


Figure 11: An excerpt of the product-line model for FMC subsea systems representing the topmost components and their relationships.

The base model serves as the context for the variability model. It must capture all the configurable elements of the system (i.e., configurable classes and their configurable properties), hierarchies that make configurable elements accessible from the topmost element, and information required for supporting configuration of the configurable properties.

For example, DeviceController is a software class that can be configured to operate in one of the two modes, normal and maintenance. To model this, in the software view we create a UML class DeviceController, with an attribute, mode, to represent its operation mode. This attribute should be captured in the model since it is a configurable property. In addition, to support configuring this property, we need to create a UML Enumeration, OperationMode, with literals normal and maintenance. This design is shown in Figure 12.

In the base model of an ICS family, all the configurable software and hardware classes, or one of their superclasses must be reachable from the topmost element of the system. This is necessary because configuration is done in a top-down manner, where the configuration engineer starts from the higher-level components, configures them, and proceeds to their subcomponents in the hierarchy.

5.4.1. Software sub-view

Software engineers are typically responsible for creating the software model. The software model contains software classes and their relationships. In the following, we describe the notation and guidelines for creating software models of ICS families.

Modeling notation

UML classes should be used to model software classes. UML generalization relationships are used to create taxonomies of software classes, while UML composition associations are used to model decomposition hierarchies containing whole/part relationships.

Modeling guidelines

The software model provides a decomposition hierarchy of software classes. Any class or software concept that fits into at least one of the criteria in the following list should be captured in the software model:

1. A class (e.g., PressureTankRegulatorSw in Figure 12) that directly introduces variability, for example through one of its attributes (e.g., engUnit). These are configurable classes and must be modeled to support configuration.
2. A class (e.g., SemApp in Figure 12) that contains other configurable classes (e.g., PressureTankRegulatorSw). These composite classes, even if not themselves directly configurable, should be modeled as part of the decomposition hierarchy to make the lower-level configurable classes accessible from the topmost element in the hierarchy.
3. A class (e.g., DeviceController in Figure 12) that has configurable subclasses (e.g., PressureTankRegulatorSw). Such generic superclasses should be modeled as abstract classes. Taxonomies of software classes, in addition to supporting the reuse of common features modeled in the generic superclass, provide a variability modeling mechanism as described in Section 5.6.
4. A class (e.g., UpdateMux in Figure 12) that is used in the specification of a constraint (e.g., ControllerConnections in Section 5.5) restricting a configurable class (e.g., DeviceController in Figure 12).

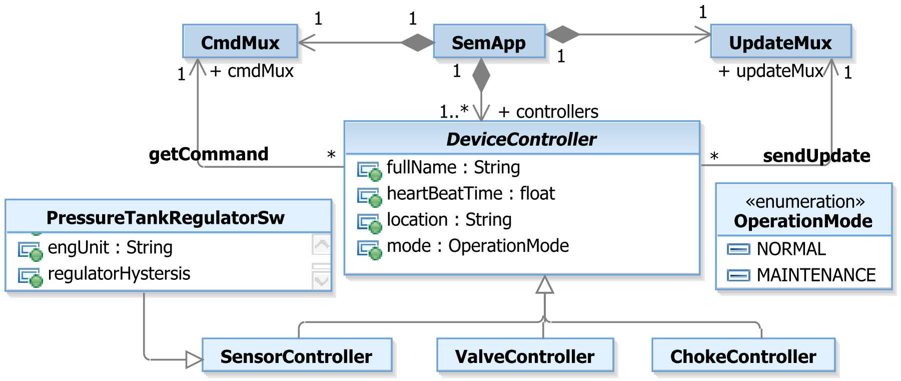


Figure 12: Decomposition of the SEM application into its subcomponents.

Figure 12 is a class diagram in the software sub-view. This class diagram describes the structure of the software class SemApp. The SemApp software is mainly composed of a number of DeviceControllers. Each instance of the DeviceController class is responsible for controlling one electrical or mechanical device. The three subclasses of DeviceController, i.e., ValveController, SensorController, and ChokeController provide the basic functionality for controlling three basic types of devices: valves, sensors, and chokes, respectively. PressureTankRegulatorSw is a subclass of SensorController and is responsible for controlling a special device called pressure tank regulator. This class has a number of properties and attributes that can be different from one system to another. As shown in the class diagram, each instance of DeviceController communicates with a CmdMux (command mux) and an UpdateMux. These two classes are used to enable the interaction with the McsApp software, which is not shown in this class diagram. Note that CmdMux and UpdateMux do not introduce any variability but are included in the software model since they affect the configuration of instances of DeviceController as specified in the OCL constraint ControllerConnections given in Section 5.5.

5.4.2. Hardware sub-views

The hardware sub-view is typically created by mechanical and electrical engineers. However, since the SimPL methodology does not require modeling of all the hardware technical details, other engineers, such as software or system engineers, may also construct this sub-view.

Modeling notation

The hardware model contains mechanical and electrical components and devices, and the relationships between them. As mentioned in Section 5.2, we use UML classes stereotyped by a MARTE stereotype to distinguish hardware classes. «HwComputingResource» is a MARTE stereotype that denotes an active execution resource. We use this stereotype to distinguish those electrical hardware components on which software is deployed. The MARTE stereotype «HwDevice» denotes auxiliary resources that interact with the environment to expand the functionality of the hardware platform. Examples of «HwDevice» are sensors, actuators, power supplies, etc. We use «HwDevice» to distinguish those hardware devices that are controlled by, or in general interact with, software. Other hardware classes that represent hardware components that physically contain other devices and execution platforms are distinguished using the MARTE stereotype «HwComponent», which denotes a generic physical component that can be refined into a variety of subcomponents.

A composition association in the hardware sub-view connecting classes stereotyped by the above mentioned stereotypes indicates a physical containment of a component in another. A generalization relationship in this sub-view indicates a classification of hardware components or devices. A class may have a set of properties, which can either be configurable or not. In addition to these properties, elements in the hardware sub-view can be characterized by the attributes of the MARTE stereotypes (e.g., «HwComputingResource») applied to them. For example, «HwComputingResource» has an attribute named op_Frequencies, which specifies the range of supported frequencies.

Modeling guidelines

The hardware model should provide a containment hierarchy that is complete with respect to the following criteria:

1. The hierarchy should contain all the hardware computing resources that have a configurable software class deployed to them. For example in Figure 13, SEM is a hardware computing resource modeled as part of the electrical hardware sub-view, since it has the configurable software SemApp deployed on it. Note that SemApp is modeled as a UML class in the software model.
2. The hierarchy should contain all the hardware (both mechanical and electrical) devices and instruments that are controlled by software. For example in Figure 13, Sensor, Choke, and Valve are hardware devices that are controlled by software classes and therefore their properties affect the operation of the software, which should be configured accordingly.
3. Modeling a component or a device requires the component or system that physically contains it to be modeled as part of the hierarchy. This is required because, as mentioned earlier, to support configuration and product derivation all configurable classes should be modeled in the hierarchy and be

accessible from the topmost element. For example in Figure 13, XmasTree is a mechanical component that physically contains the instruments mentioned above and should, therefore, be captured in a model in the hardware sub-view, even though it itself contains no software. Note that XmasTree is a subcomponent of SubseaField, but this decomposition is not shown in Figure 13. Modeling XmasTree in the hardware model allows accessing Sensor, Choke, and Valve when traversing the hardware model starting from the topmost element in the system design view.

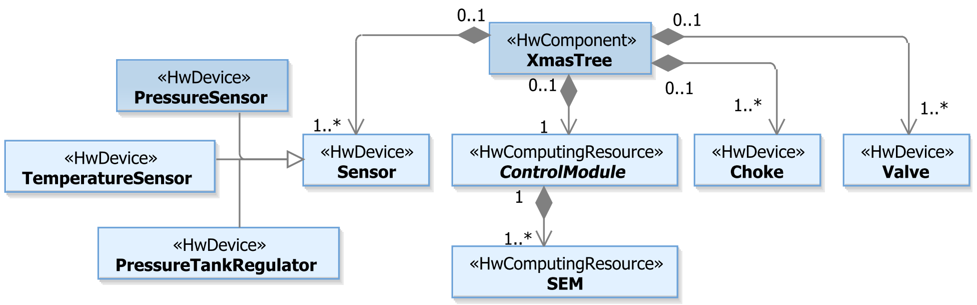

«HwComponent» XmasTree
«HwDevice» PressureSensor
«HwDevice» TemperatureSensor
«HwDevice» PressureTankRegulator
«HwDevice» Sensor
«HwComputingResource» ControlModule
«HwDevice» Choke
«HwDevice» Valve
«HwComputingResource» SEM
0..1
0..1
0..1
0..1
1..*
1
1..*
1..*
1
1..*

Figure 13: A model in the hardware sub-view showing the decomposition of XmasTree into its subcomponents.

An example hardware model is shown in Figure 13. This model is an excerpt of the generic model constructed for the FMC case study and depicts the decomposition of the «HwComponent» XmasTree into its subcomponents: ControlModule, Sensor, Choke, and Valve. This decomposition is modeled using UML composition associations. This figure also shows another step of decomposition for ControlModule. ControlModule and SEM are electrical components that can have software deployed and are, therefore, stereotyped by MARTE «HwComputingResource». As mentioned earlier taxonomies of hardware components and devices can be modeled using UML generalization relationships. A taxonomy of Sensors is shown in Figure 13 containing PressureSensor, TemperatureSensor, and PressureTankRegulator as subtypes of Sensor.

5.4.3. Allocation sub-view

The allocation model pairs software and hardware classes indicating that instances of the software class can be deployed to instances of the hardware class. Note that, the actual deployment of an instance of a software class to a computing resource is required to be captured as part of the configuration using instance-based models (Section 6). The allocation sub-view is a mixed view in the sense that it has to import model elements from both software and hardware sub-views. Software and electrical engineers are responsible for creating this sub-view.

Modeling notation

We use the MARTE stereotype «Assign» to model the deployment, allocation, or binding of a structure (e.g., software class) in the software sub-view to a resource (e.g., a hardware component) in the hardware sub-view.

Modeling guidelines

Allocation, in MARTE, is a domain concept used to associate individual application elements to individual execution platform elements. The MARTE stereotype «Assign» realizes the concept of allocation, and is applicable to UML comments. To model the deployment of a software component onto a hardware component, first, a class representing the software component and a class representing the electrical component where the software component is deployed should be imported into the allocation sub-view. Then, we use the MARTE stereotype «Assign» to model the deployment.

Figure 14 shows an example. In this figure we have used the graphical notation suggested in the MARTE specification to visualize two deployments: the deployment of software McsApp class to electrical component MCS, and the deployment of software class SemApp to electrical component SEM.

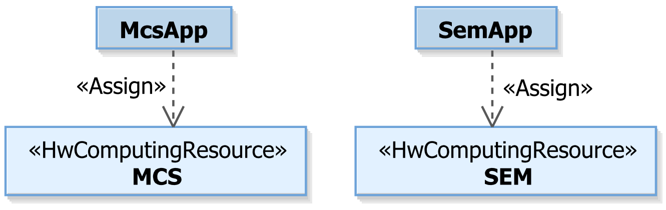


Figure 14: An example of allocation sub-view.

5.4.4. Example

Figure 15 shows a small excerpt of the FMC model. This class diagram shows the system design view, including class FMCSystem and some of its subcomponents, namely SubseaField and SemApp, which are captured in separate models. The diagram also partially shows how subcomponents and parts of SubseaField and SemApp are organized in software and hardware models. Note that in this diagram those packages that have a viewpoint stereotype applied represent views, while the other packages are used to provide a finer-grained organization of the model.

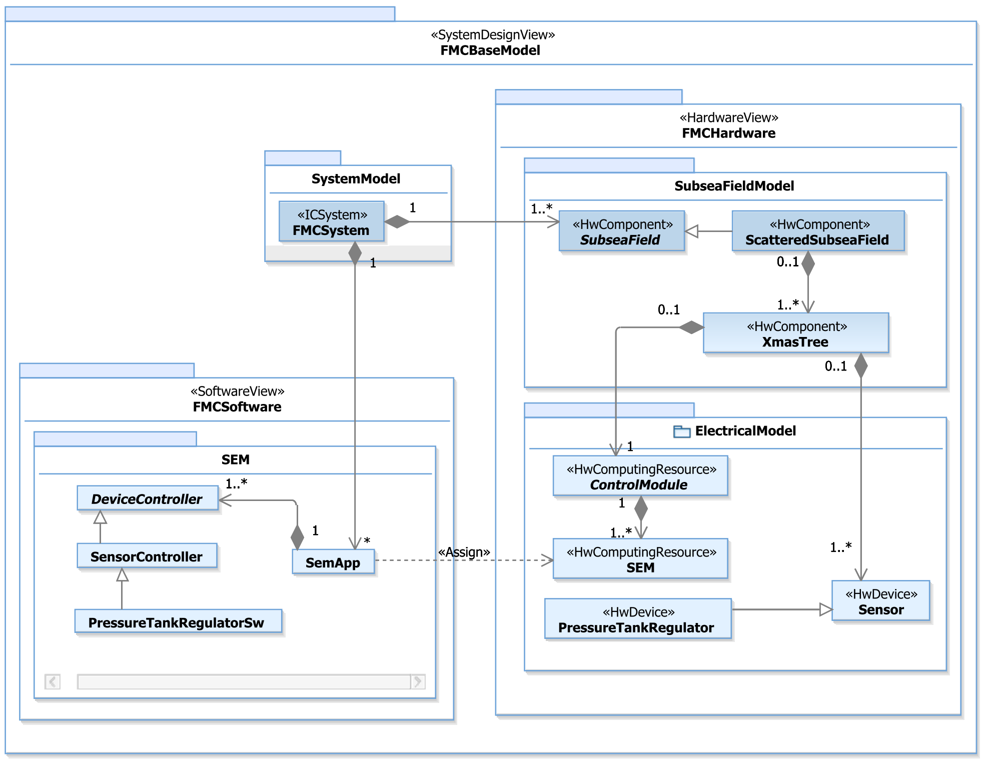

«SystemDesignView» FMCBaseModel
SystemModel
«ICSystem» FMCSystem
1
1
«HardwareView» FMCHardware
SubseaFieldModel
1..*
«HwComponent» SubseaField
«HwComponent» ScatteredSubseaField
0..1
1..*
0..1
«HwComponent» XmasTree
0..1
ElectricalModel
1
«HwComputingResource» ControlModule
1
1..*
«HwComputingResource» SEM
1..*
«HwDevice» PressureTankRegulator
«HwDevice» Sensor
«SoftwareView» FMCSoftware
SEM
DeviceController
1..*
1
SensorController
SemApp
*
PressureTankRegulatorSw
«Assign»

Figure 15: An excerpt of the FMC family model created by following SimPL.

As shown in the diagram, SubseaField is a mechanical component in the hardware model (i.e., FMCHardware). SemApp on the other hand is owned by a package (i.e., SEM) in the software model (i.e., FMC-Software).

PressureTankRegulatorSw is a software class, which is an indirect subtype of DeviceController. Like any instance of DeviceController, any instance of the class PressureTankRegulatorSw is owned by an instance of the SemApp software. This fact can be derived from the composition association between DeviceController and SemApp and the fact that PressureTankRegulatorSw is a subtype of DeviceController.

5.5. Constraint modeling

Classes and relationships are insufficient for modeling all the necessary information about a system, and, therefore, UML models are usually augmented with a number of constraints, each expressing restrictions or conditions on a UML element to declare some of its semantics or to define its dependencies on other model elements. In the SimPL methodology, these constraints can be added to the base model at any point during the modeling process. We use the Object Constraint Language (OCL) [2], which is associated with UML, to precisely express constraints in our context.

Constraints in the domain of families of ICSs can be classified, according to their *scope*, into two categories: *domain-specific constraints* and *application-specific constraints*. Domain-specific constraints are the constraints that hold for all members of an ICS family, and therefore, should be captured in the base model of the ICSs family. An example of such a constraint is that, in Figure 12, all instances of DeviceController in an instance of SemApp should be associated and connected to an instance of UpdateMux and an instance of CmdMux that are owned by the same instance of SemApp. This constraint is expressed using an OCL constraint named ControllerConnections as follows, in the software view of the system:

```
context SemApp inv ControllerConnections
controllers->forAll(c : DeviceController |
                c.updateMux = self.updateMux and
                c.cmdMux = self.cmdMux)
```

Application-specific constraints, on the other hand, are the constraints that are applied only to a specific member of the family. These constraints should be separately captured for each member of the family, and should be supplied to the configuration tool as part of the configuration data. For example, in a specific product instance of FMC subsea systems, we have the constraint that says, for safety purposes, all control modules should include two SEMs, SEM_A and SEM_B. Also, the SemApp software deployed to SEM_A should be connected to and interact with the SemApp software deployed to SEM_B.

Domain-specific constraints can be categorized, according to the sub-views they involve, into *intra-view constraints* and *cross-view constraints*. Intra-view constraints are those constraints that involve model elements from only one sub-view. For example, the ControllerConnections constraint described above constrains only elements from the software sub-view and, therefore, falls into this category of constraints.

Cross-view constraints, on the other hand, constrain elements from several sub-views. Cross-view constraints specify consistency rules between hardware and software models. For example, a consistency rule that specifies that the number of hardware devices (Sensors, Chokes, and Valves) controlled by an instance of SEM should be equal to the number of instances of DeviceController owned by the instance of SemApp deployed to that instance of SEM is modeled using the following OCL constraint:

```
context SemApp inv controllersNumebr
controllers->size() = sEM.chokes->size()
                       + sEM.valves->size()
                       + sEM.sensors->size()
```

5.6. Variability view

The variability model must capture all static variability points (i.e., variability points that are resolved prior to start of execution) [7], including both structural and behavioral variabilities. However, variabilities in the behavior of software are usually handled through parameterizing the generic code base[8]. As a result, it is sufficient to rely only on structural modeling constructs to model variabilities.

In this section we first present a taxonomy of variabilities in Section 5.6.1. This taxonomy defines all types of variabilities that exist in ICS families. In Section 5.6.1, we also discuss how inherent UML features are used to model configurable elements in the base model to support modeling different types of variabilities. Then, in Section 5.6.2, we describe how we take advantage of inherent UML features to capture and organize, in the variability model, variability points pointing to their corresponding configurable elements in the base model.

5.6.1. Taxonomy of variability types

Variabilities are those aspects in the architecture, design, or implementation of a family of products that can vary from one product to another. These aspects should be captured in the base model provided for the product family. UML classes and relationships between them are used to model the structure of a system or software, as shown in the example models presented in Section 5.4. The following is a classification of variabilities existing in the domain of ICSs families:

1. **Cardinality variability**: In a system or software, the number of instances of a certain type can vary from one product to another. We use UML properties (either aggregated or composite) and their multiplicities to model this type of variability. For example, in the software model given in Figure 12, the number of DeviceControllers contained by an instance of SemApp can vary from one subsea production field to another, hence resulting in different products. As shown in the class diagram in Figure 12, the 1..* multiplicity on the composition association connecting SemApp to DeviceController is used to capture this variability. Note that not all such multiplicities represent a variability. Multiplicities that do not introduce variability correspond to cardinalities that dynamically change during runtime, for example, as a result of a change in the state of a subsystem or component. Such multiplicities are common in software models, and correspond to sets of objects that are dynamically created and destroyed during runtime. Variability points in the variability models are required to distinguish between the multiplicities that introduce variability and the ones that do not.

[8]There are several ways to do this. For example, one type of variability in behavior can be captured using classes with polymorphic behavior. In this case, a parameter in the code base can be used to indicate which class (and, therefore, which behavior) should be used in a particular product instance. A parameter can, as well, be used in a switch statement or in an if-block to indicate which branches should be executed in a particular product instance.

2. **Attribute variability**: The value of a configurable attribute of a class can vary from one instance of the class to another. For example, in Figure 12, the value of attribute engUnit of class PressureTankRegulatorSw might be different for each instance of PressureTankRegulatorSw and therefore it represents a variability. Note that not all attributes represent variabilities. Only those attributes that are referred by a variability point represent variabilities and are configurable. A configurable attribute, when resolved for an instance of a class in the context of a specific product instance, keeps its value during the whole lifetime of that instance. In contrast, the value of a non-configurable attribute changes during the lifetime of the owning object as the system operates.
3. **Topology variability**: The structural topology of the system or a component, i.e, the connections between instances of its contained classes, can vary from one product to another. Topological structures are typically represented through instance-based models (as opposed to class-based models), where runtime instances (e.g., objects, roles) and their relationships (e.g., links, connectors) are modeled. For example consider the class diagram in Figure 16-(a), which shows class SEM with a self-association. This class diagram is a part of the FMC product-family model. relativeSems is a configurable property of SEM. Figures 16-(b) and 16-(c) show excerpts of two product models both having five instances of SEM, but having two different topological structures. The topological difference is due to the fact that the configurable property relativeSems can be configured differently for each SEM instance.

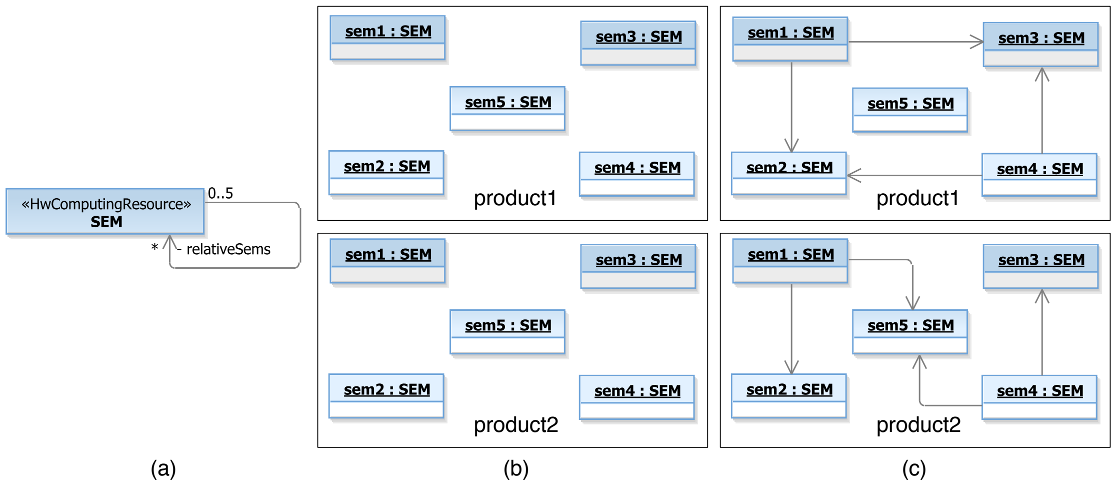


Figure 16: An example of topology variability.

4. **Type variability**: For each instance, its concrete type can vary from one product to another. We use UML generalizations to model taxonomies of types. We type instances using abstract classes for cases where the concrete type for a corresponding instance is defined at configuration time. For example, in Figure 12, the type of an instance of DeviceController owned by an instance of SemApp can vary among SensorController, ChokeController, and ValveController. The concrete type of an instance specifies its detailed implementation, particularly in terms of:
   (a) The number and type of its parts (e.g., subcomponents and subsystems).
   (b) The internal structure of its parts, which can match a particular topological pattern. Each topological pattern can be associated with a separate structured class representing a concrete subtype of a common superclass.
   (c) Its behaviors, which can be modeled using behavioral modeling capabilities of UML (e.g., operations, state machines, or interactions).

5.6.2. Variability modeling using UML templates

The variability model in the SimPL methodology specifies and organizes all the variability points. Each variability point refers to (or points to) a configurable element in the base model.

Modeling notation

UML concepts, such as composition/aggregation, generalizations/specialization, and parameterization, are the primary mechanisms for specifying genericity (i.e., commonalities and variabilities in the context of variability modeling) in the base model. For the variability model, we primarily use the UML template concept. A UML *template* is a model element in which one or more of its constituent parts are not fully specified. Instead, they are designated as *template parameters*, to be fully defined later when the template is formally bound. This allows different concrete model elements to be generated from a single template specification. For example, in a class definition, the type of one of its attributes may be designated as a parameter. Binding is achieved by substituting concrete values for the template parameters, such as allocating a concrete type for a parameterized attribute.

In addition to UML templates, we use three stereotypes (i.e., «ConfigurationUnit», «RelatedConfigUnit», and «Inherit») from SimPL to model variabilities. The use of UML templates and these SimPL stereotypes is explained through the modeling guidelines described below.

Modeling guidelines

The variability model is a collection of configuration units, which as mentioned in Section 5.2 group variability points according to their origin. To create the variability model one should start by creating such configuration units in the variability view. A configuration unit is a template package stereotyped by the SimPL stereotype «ConfigurationUnit», and associated to its origin class using a dependency stereotyped by «RelatedConfigUnit». Figure 17 shows a simple example. Note that these two stereotypes are required to ease the process of traversing the variability model and the base model during configuration.

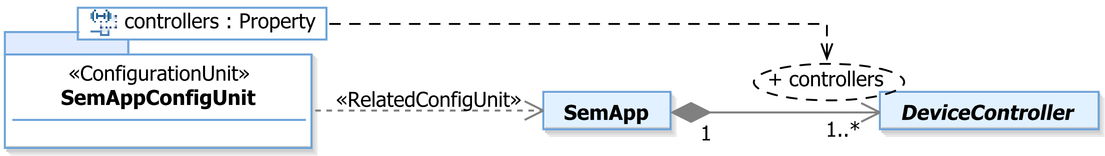


Figure 17: An example of UML template concepts and the use of «ConfigurationUnit» and «RelatedConfigUnit».

In SimPL, we use a simple form of UML *package templates* (i.e., the template specification in a template package [3]) for specifying variability points in a configuration unit. In this case, the parameters in a package template are *references* to configurable properties in the origin class. In Figure 17-(a), the package SemAppConfigUnit is a template with a parameter named controllers. This parameter is typed by Property, which means that it designates a model element that represents a kind of UML property and, hence, can only be substituted with concrete values that represent UML properties.

The dashed line in Figure 17 shows the traceability link that relates the controllers template parameter to its related property in the origin class SemApp. Note that the configurable property in the origin class is an association end also called controllers. According to the UML metamodel [3], such a traceability link between a template parameter and the configurable property in the base model is inherently maintained by UML in a semantically correct UML model.

In general, in UML, parameters of a package template can refer to different types of model elements (e.g., class, property, operation). However, in our method for modeling variabilities, we found it sufficient for a package template representing a configuration unit to have only parameters referring to UML properties. Note that a UML property has several attributes characterizing it. Two key attributes of a UML property are *multiplicity* and *type*. A template parameter referring to a UML property with range multiplicity (e.g., multiplicities such as *, 1..*, 0..10) represents a cardinality variability. To resolve this variability, one should assign a fixed value to the multiplicity. On the other hand, a template parameter referring to a UML property typed by an abstract class or a class that has several subclasses represents a type variability. To resolve such a variability, one should select among the concrete subclasses of the type of the referred UML property. A template parameter referring to an attribute of a class represents an attribute variability. To resolve this variability, an appropriate value should be assigned to that attribute. By an appropriate value we mean a value that is a valid instance of the type of the configurable attribute, and does not result in the violation of any OCL constraint defined in the model. As mentioned in Section 5.6.1 a unidirectional association connecting two classes in the base model is used to model the topology variability. In this approach, the template parameter representing the related variability point should point to the association end of this unidirectional association. To resolve such a topology variability, one should create appropriate association instances (i.e., links) between instances of the two involved classes.

The combination of UML package and template provides an elegant mechanism to support variability modeling with the following advantages. First, the solution makes use of a single unified modeling language (i.e., UML) to model both the base model and the variability model. Second, the variability model is loosely connected to its base model and therefore the evolution of the variability model and the base model can be independent to a certain extent. For example, changes to the base model that do not introduce the addition, deletion or modification of the variability points specified in the corresponding package template of the variability model have no impact on the variability model. Another advantage of the loose connection between base and variability models is that system modelers need not be constrained in how they choose to model the system just to accommodate the variability model, which lessens their concerns and gives them flexibility.

The relationships that exists between classes in the base model affects the structure of the variability model. Consider two classes A and B, where B is a subclass of A. Also assume both classes introduce some variability, for example through some of their properties. Therefore, in the variability model we need two configuration units, Ca and Cb, respectively grouping the variability points introduced by A and B. Since any instance of B is also an instance of A, to configure an instance of B, in addition to resolving the variability points modeled in Cb, we need to resolve all the variability points in Ca since they refer to some properties inherited by B. In fact, the configuration unit Cb inherits variability points from Ca. In order to explicitly model such a dependency between configuration units, we use a unidirectional UML dependency connecting the sub-configuration-unit to the super-configuration-unit. This dependency is specified via the «Inherit» stereotype from the SimPL profile.

In short, to create the variability model, for each configurable class in the base model, we create a template package in the variability view, and stereotype it as a «ConfigurationUnit». We add template parameters to the definition of the template package to represent the variability points introduced by the origin class. Finally, we connect the template package to the origin class using a dependency stereotyped by «RelatedConfigUnit». «Inherit» relationships between template packages in the variability view will be added as we proceed.

Figure 18 is an extension to the example in Figure 15, and shows a class diagram in the variability view. This class diagram shows seven configuration units (i.e., FMCSystemConfigurationUnit, SemAppConfigUnit, DeviceControllerConfigUnit, PressureTankRegulatorSwConfigUnit, ScatteredSubseaField-

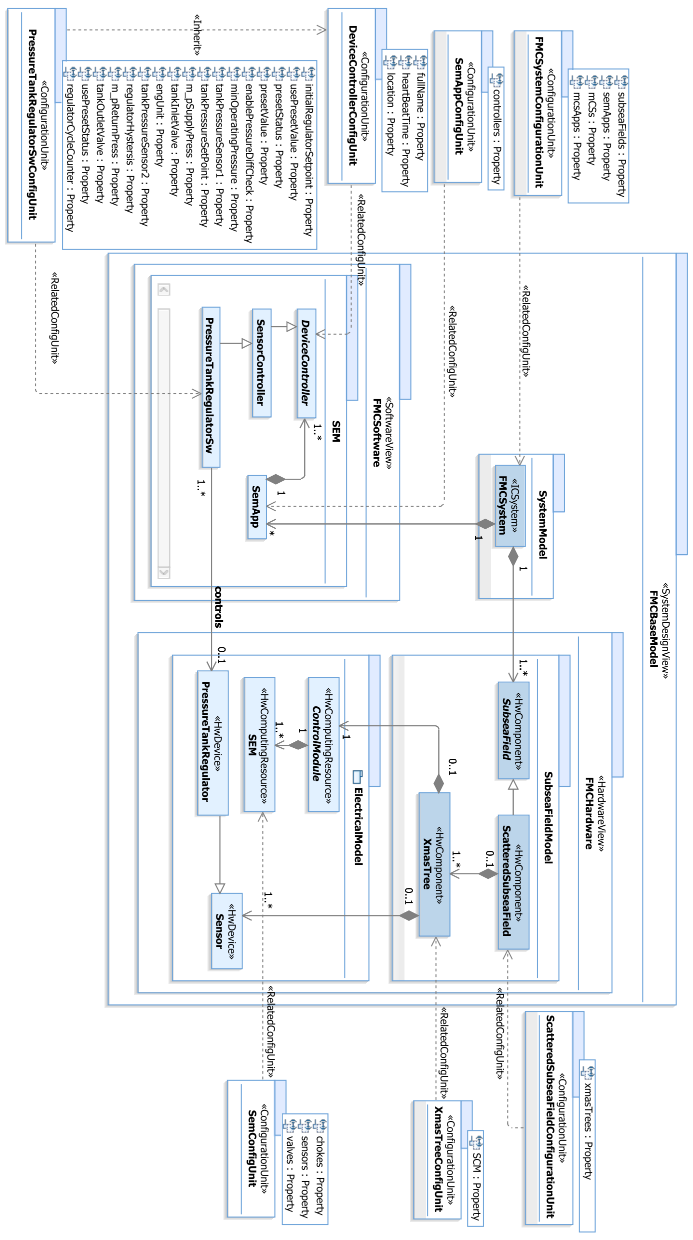


Figure 18: Extension to the class diagram in Figure 15, showing the variability points and configuration units.

ConfigUnit, XmasTreeConfigUnit, and SemConfigUnit) and their origin classes (i.e., FMCSystem, SemApp, DeviceController, PressureTankRegulatorSw, ScatteredSubseaField, XmasTree, and SEM). The configuration units are modeled as template packages stereotyped by «ConfigurationUnit», and are related to their origin classes using «RelatedConfigUnit» dependencies. Note that the origin classes are imported from the base model.

FMCSystemConfigurationUnit is a template package with four template parameters, i.e., subseaFields, SemApp, mCS, and mcsApp, representing four variability points referring to four configurable properties in the origin class, FMCSystem (the topmost element of the system). The variability point subseaFields refers to a configurable property of FMCSystem modeled using a composition association. To resolve this variability point, one should first determine the number of instances of SubseaField owned by the instance of FMCSystem, and for each instance of SubseaField its exact type should be identified. The latter is necessary since SubseaField is an abstract class and has two subclasses (i.e., ScatteredSubseaField and TemplatedSubseaField), from which the exact type of an instance should be selected.

The template parameter heartBeatTime in DeviceControllerConfigUnit refers to the configurable attribute of the abstract class DeviceController. This attribute is public and is inherited by all subclasses (e.g., PressureTankRegulatorSw) of DeviceController. As a result, there is an «Inherit» dependency between the sub-configuration-unit PressureTankRegulatorSwConfigUnit and the super-configuration-unit DeviceControllerConfigUnit. As shown in the class diagram, class PressureTankRegulatorSwConfigUnit has itself a number of template parameters referring to the configurable attributes (e.g., initialRegulatorSetpoint) of class PressureTankRegulatorSw.

6. Product configuration

The second step in Figure 2 is the semi-automated configuration step. During the semi-automated configuration step, a product specification is created from a generic model of a product family (i.e., SimPL model) and the configuration decisions provided by the user. To complete the picture, we briefly explain here the architecture-level configuration of ICS families in a model-based context. More details about this configuration approach and a solution for (partially) automating it are provided in [8].

As opposed to generic models, which are class-based models, product specifications are instance-based models. A software product specification is a collection of instances of configurable software classes defined in the SimPL model of the product family. Throughout the configuration process, configuration engineers provide information required for each instance by assigning values to its configurable parameters. This may include creating new instances and new connections between instances. Each value assignment in this context represents a configuration decision.

Figure 19 is a UML object diagram representing the product specification of a simple product instance of an FMC product family presented in Figures 11 through 18. This product specification is created by making 15 different configuration decisions. The model in Figure 19 only shows a greatly simplified example product, which is similar but much smaller than real products. In terms of scalability, drawing such diagrams is not likely to be practical for a complete system, and configuration engineers are not expected to create such diagrams in the semi-automated configuration step. Instead, they provide configuration decisions (e.g., via a domain-specific front-end tool) from which such diagrams can be created.

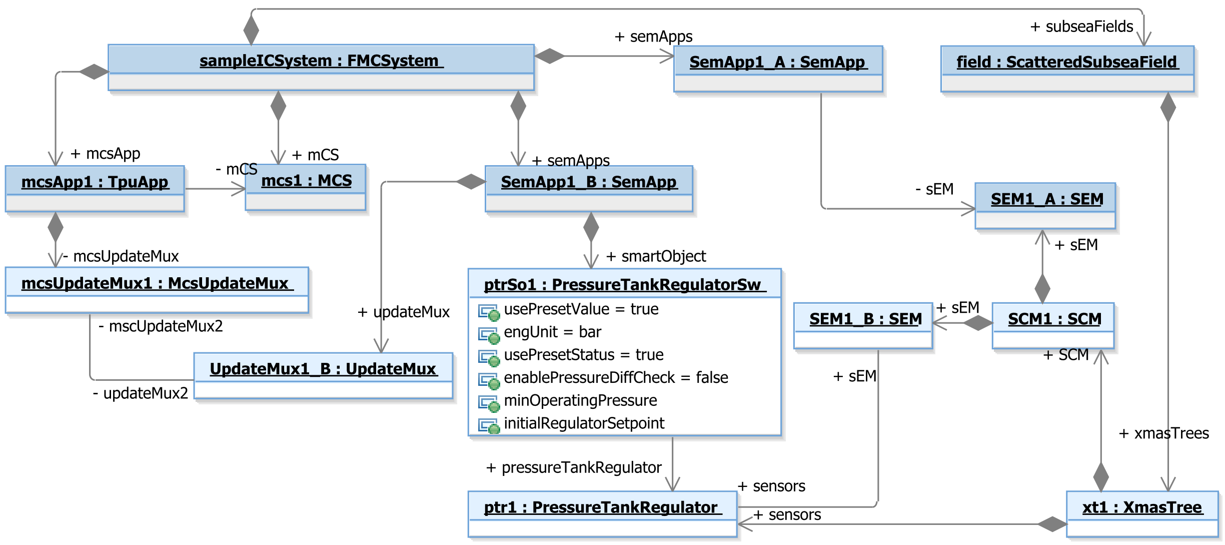


Figure 19: An example product specification.

Ensuring the correctness and consistency of the software configuration, especially with respect to the hardware configuration, requires providing the configuration engine with information about the underlying hardware configuration. Note that (1) the software configuration engineer can provide the required information about the hardware configuration because the hardware configuration is, usually, one of the inputs to the software configuration process, and (2) since the software configuration closely follows the hardware configuration, this additional information about the hardware does not result in configuration overhead. Instead, information about the hardware configuration can be used to automatically create a skeleton for the software configuration.

7. Evaluation and discussion

In this section, we evaluate the SimPL methodology by assessing the extent to which it fulfills the modeling requirements specified in Section 4.3. Recall from Figure 2 that the SimPL methodology is the first step towards a solution for the configuration challenges in the ICS domain. Details of the automation support for configuration, which is the second step of our configuration solution in Figure 2, and an evaluation of such an automation support are presented in [8] and [9]. In particular, results reported in [8] show that the proposed configuration solution can provide the automation described in Section 4.1. In addition, an analysis of the applicability of the configuration solution for the purpose of automated configuration reuse is presented in [9]. Results of that work show that, using our approach, more than 60% of configuration decisions for the subjects in our experiment can be automatically derived through the reuse of configuration data. Note that both approaches presented in [8] and [9] are based on input models created by following the SimPL methodology. In the remainder of this section, we focus on the evaluation of the SimPL methodology.

7.1. Evaluation of the SimPL methodology: an industrial case study

To evaluate the ability of the SimPL methodology in fulfilling the modeling requirements (Section 4.3), we applied it to model an industrial case study. For this purpose, we chose a family of FMC subsea oil production systems, which is representative in terms of the characteristics of ICS families described in Section 3.1. In the remainder of this section, we refer to this family as SubseaFamily09. We started by studying (1) the software code base of SubseaFamily09 to extract software configurable classes, and (2) the documentation of the underlying hardware to identify different types of hardware configurable components, and hardware/software dependencies. Both software and hardware in SubseaFamily09 are hierarchical. There are 53 configurable classes in the lowest level of the software hierarchy, and more that 60 hardware component types in the system[9]. The main types of hardware components in a subsea oil production system are, as described in Section 3.1, Xmas trees, subsea electronic modules (SEM), and electrical and mechanical devices. The hierarchy of software classes contains device controllers and specifies the architecture of the software application that is deployed on SEM. Configurable classes in the software hierarchy introduce about 350 variability points, and hardware configurable component types introduce about 100 variability points. Among the variability points in software, 205 are attribute variabilities (70 boolean variables and 135 variables of type integer or enumeration), 41 are cardinality variabilities, 51 are type variabilities, and 50 are topology variabilities.

In addition, we studied two representative products derived from SubseaFamily09. Products that FMC produces typically consist of thousands of configurable components and their configuration involves assigning values to tens of thousands of configurable parameters. Table 1 gives an overview of the two products that we investigated.

Table 1: An overview of the two real products derived from SubseaFamily09.

	# XmasTrees	**# SEMs**	**# Devices**	**# Config. params**
Product_1	9	18 (9 twin SEMs)	2360	29796
Product_2	14	28 (14 twin SEMs)	5072	56124

Based on our studies of the product family and its representative products, we employed the SimPL methodology to create a product-family model for SubseaFamily09. The product-family model that we created is not intended to exhaustively capture all the commonalities and variabilities of the family. Instead, the intention was to create a model possessing all the different kinds of modeling features from the SimPL methodology (e.g., all types of variabilities and all types of dependencies between them). To create such a model, based on our knowledge of the system (e.g., configurable classes, variabilities they introduce, and the dependencies between them), we selected and modeled a number of software configurable classes, and related hardware components. We used IBM RSA 8 [4] to create the SimPL model of SubseaFamily09. RSA was selected as the modeling environment because it incorporates an accurate and complete realization of the standard UML metamodel, and, in addition, it allows creation of UML profiles. We have implemented the SimPL profile (presented in Section 5) in RSA, and applied it to the models that we created for SubseaFamily09.

To validate the models, we discussed them in two workshop sessions with engineers at FMC and revised the models according to the feedback we got. The final revision of the model was approved by FMC engineers.

[9]We do not know the exact number of all hardware components as the documents that we studied were neither up-to-date nor complete.

Example models used in Section 5 to illustrate our methodology are sanitized excerpts of the approved model that we created for SubseaFamily09.

Table 2: Characteristics of the product-family model created for SubseaFamily09.

#Views	#Diagrams	#Classes	#Config. units	#Vars	#Constraints
5	17	71	22	109	16

Table 2 gives the characteristics of the resulting product-family model, which in total has five views and sub-views and is visualized using 17 class diagrams. The model contains a total of 71 classes, including 46 software classes, 24 hardware classes, and a class representing the topmost element, FMCSystem. The software model is an abstraction of the family's code base and contains a hierarchy of both configurable and non-configurable software classes, their attributes, and their relationships. The hardware model captures a subset of devices (i.e., only those devices that are controlled by software classes captured in the software model), their attributes, and the supporting containment and taxonomic hierarchies. The result is a hardware model with 24 hardware components and devices, including 11 computing resources. Two types of relationships between the software and hardware classes (i.e., allocation of software to computing hardware and software controlling hardware) are captured in the allocation model. The variability model contains 22 configuration units that are modeled using 22 template packages stereotyped by «ConfigurationUnit». These configuration units correspond to 22 configurable classes (i.e., origin classes of the configuration units) in software and hardware models. A total of 109 variability points – covering all types of variabilities introduced in Section 5.6 – were modeled and organized in these configuration units. In addition, a total of 34 «Inherit» and «RelatedConfigUnit» dependencies were created to complete the variability model. A total of 16 OCL constraints were defined to model domain specific constraints, in particular to model dependencies between the elements in hardware and software models. Some of these OCL expressions are relatively complex. In our case study, they consist of nested select statements, comparisons of object collections, and let clauses that define up to six related auxiliary sets. The set of OCL constraints that we have defined in our case study are presented in Appendix D.

In short, SimPL satisfies all the modeling requirements for the subject of our case study, which is a representative ICS family. More specifically:

- **Base modeling.** Three of the requirements in Section 4.3, namely software modeling (Req2), hardware modeling (Req3), and modeling software-hardware dependencies (Req4), concern base models. The structural modeling concepts of UML and the four stereotypes imported from MARTE were sufficient for fulfilling these modeling requirements in our case study. In particular, UML, as an appropriate object-oriented modeling language, provides all the required constructs, such as classes, properties, enumerations, and relationships to create a concise abstraction of the software code base of SubseaFamily09, thereby fulfilling the software modeling requirement (Req2). Regarding hardware modeling (Req3), our experience with SubseaFamily09 showed that the four MARTE stereotypes imported to the SimPL profile were sufficient for modeling various types of hardware components that we usually find in the domain. Different types of relationships that can be defined between UML classes provide us with the necessary constructs for defining generalization and composition hierarchies in software and hardware models, and defining relationships between component types. Using UML classes for modeling both hardware and software allows us to easily model software-hardware dependencies, therefore fulfilling Req4. Moreover, we found OCL expressive enough to capture the constraints in the base model. However, some OCL expressions turned out to be complex and difficult to express correctly.

- **Variability modeling.** Two major requirements presented in Section 4.3 describe the need for comprehensively modeling all types of variability points (Req1) and tracing them back to their corresponding model elements in software and hardware models (Req5). In our experience, the template modeling mechanism of UML provides the required constructs for comprehensively modeling all types of variability points (i.e., attribute, cardinality, type, and topology) and tracing them to the elements in the base model. This is because every structural element of UML used in the SimPL profile is parameterable and can be pointed to by a template parameter in a template (which, in our approach, is a package template). In particular, attribute, cardinality, topology, and type variabilities are, respectively, defined using parameterable attributes, multiplicities, association ends, and types with their corresponding generalization hierarchies.

- **Hierarchical organization of variability points.** The last requirement in Section 4.3 expresses the need for the hierarchical organization of variability points similar to that of software and hardware component hierarchies. One challenge in fulfilling this requirement is how to model nested configurable component types and their variability points. Packages in UML can be used to organize model elements into nested hierarchies. This seems to be useful for modeling nested structures where a configurable component is contained by another configurable component. However, we found it too complex to use nested template packages for this purpose[10]. Instead, we model configuration units using a flat organization of template packages. A hierarchal representation of variability points that is easy to understand for configuration engineers (Req6 in Section 4.3) can be generated from such template packages and the composition associations between their origin classes.

7.2. Discussion

The experiments described in [8], and the case study reported in Section 7.1 evaluate the ability of our model-based configuration approach with respect to the first two characteristics (i.e., automation and completeness) of a configuration solution described in Section 4.1. In short, these evaluations show that our approach can provide the required automation. Moreover, our results show that there are no technical challenges regarding collecting and representing all the configuration decisions required for deriving executable software products, therefore ensuring the completeness of the solution. In particular, our experience with the case study reported in Section 7.1 shows that SimPL is capable of modeling all types of variability points, which is the first step to collecting configuration decisions.

The scalability of our approach for the configuration of large products is discussed in [8]. Our initial results show that, apart from limitations in the implementation of our prototype tool, the approach can scale well for the products that we normally find in the ICS domain, i.e., products with 2000 to 5000 configurable component instances and 20,000 to 50,000 configurable parameters.

Regarding the ability of the SimPL methodology to model large-scale product families, we consider two perspectives. The technical perspective, which addresses how the notation and modeling tool support can

[10]The main challenge in using nested template packages for modeling hierarchies of configurable components arises when a configurable component type can be contained by several other component types. For example in Figure 15, XmasTree is a hardware component that can be contained by a ScatteredSubseaField, but it can also be contained by other types of subsea fields (not shown in Figure 15). Using nested template packages for modeling nested hierarchies of variability points may, thus, require creating several identical configuration units for the same configurable component type (e.g., XmasTree), in various locations in the hierarchy. Such an approach can result in scalability and maintenance issues, and, therefore, was not chosen in SimPL.

scale; and the modeler's perspective, which addresses how our approach can help modelers to handle the complexity of large-scale product families.

Modeling large-scale product families: the technical perspective

A major necessity for creating models of families of large-scale systems, such as ICS families, is the availability of notations and tools that scale well. To fulfill this need, we have based our modeling methodology on well-known industry-standards, i.e., UML and MARTE. UML is a widely accepted modeling notation, which is supported by a wide range of open-source and commercial tools. UML tools can easily support capturing hundreds or even thousands of classes and their relationships in a model, which is sufficient for creating models of ICS families.

Most UML tools support UML extension mechanisms, thereby enabling us to easily integrate additional profiles (i.e., our SimPL profile, and the MARTE profile) with standard UML to get proper tool support for the SimPL methodology. Moreover, this integration does not affect the scalability of the tool. In short, being a methodology based on widespread and mature standards supported by widely available tools makes SimPL scalable from a technical perspective.

Modeling large-scale product families: the modeler's perspective

From a modeler's perspective the major challenges in dealing with models of large-scale systems are (1) making and understanding the model, (2) navigating the model, and (3) ensuring the correctness of the model.

Apart from modeling skills, making and understanding models require knowledge about the modeling notation and using the modeling tool support. Basing our methodology on a well-known and widely used industry-standard notation helps meet this requirement.

The SimPL methodology assists model navigation by organizing the model hierarchically, and into multiple views. In addition to facilitating the navigation of the models, we expect such a modeling approach to help modelers better deal with the complexity (e.g., heterogeneity and configurability) of ICS families.

To help modelers ensure the correctness of the models, we have implemented domain-independent consistency rules (i.e., rules that ensure the consistency of different views of the system) in our SimPL profile, using a set of OCL constraints among the viewpoint stereotypes (Section 5.2). Some UML tools (e.g., IBM RSA 8 [4]) support automated verification of profiled models with respect to the semantics defined in a profile, i.e., relationships and the constraints among the stereotypes in the profile. Using this capability, and with the help of our SimPL profile, we can assist modelers throughout the modeling process to ensure, to some extent, the correctness of their models.

In short, we believe that our modeling methodology provides the required foundation for modeling large-scale product families. However, a solid evaluation and analysis of scalability of our approach, with respect to the size of ICS families, is still required. Such an evaluation requires performing field experiments with human subjects, and is left for future work.

8. Related work

A product family can be specified on three levels of abstraction: feature level, architecture level, and component implementation level [33]. Variabilities may exist in any abstraction level. Variability at one abstraction level realizes variability on higher abstraction levels (e.g., variability in the implementation of a component realizes variability in the architecture). While some approaches (e.g., [33, 32]) support modeling product families at all the three levels of abstraction, most of the work in the literature focuses on modeling product families only at one or two of these levels. For example, basic feature modeling [24] models product families only at the highest level of abstraction. In this technical report, we have proposed the SimPL methodology, which is designed for the architecture-level modeling of ICS families. In the remainder of this section, we review the existing approaches to architecture-level modeling of product families and evaluate them by assessing their ability in fulfilling the specific modeling requirements identified in Section 4.3. Our evaluation shows that, in contrast to the SimPL methodology, none of the existing approaches fulfill all of these modeling requirements.

Basic feature modeling is extended in [25, 16, 14] to enable architecture level modeling of product families. In particular, extended feature models (EFM) that allow attributes, cardinalities, references to other features, and cloning of features are, as mentioned in [14, 34], as expressive as UML class diagrams and can be used to model variabilities at the architecture level.

Alternatively, several approaches (e.g., [21, 15, 13]) have focused on combining feature models with models of architecture and design artifacts such as requirements, use cases, class diagrams or design documents. In these approaches, UML as a standard modeling language has been the preferred language for specifying the architecture of a product family.

COVAMOF [33] is a variability modeling framework that models variability in terms of variability points and dependencies. In COVAMOF, variability is modeled uniformly over different abstraction levels (e.g., features, architecture, and implementation). Variability points and dependencies are captured in a variability view that can be derived from the product family artifacts manually or automatically. COVAMOF presents a graphical notation and an XML-based textual notation (CVVL) for modeling variability points, variants, and dependencies.

Another group of product-line modeling approaches are based on introducing variabilities through distinct variability models. These models are supplementary to the base models, which do not explicitly capture variability. These approaches can be categorized on the basis of their base modeling language, variability modeling language, and on whether they combine (amalgamate) or separate the base and variability models. Base models can be developed using UML as in [32, 38], or using domain specific languages (DSLs) as in [22, 28]. In approaches that use UML to describe base models, usually UML inherent features such as templates (e.g., in [32]) or stereotypes and profiles (e.g., in [38]) are used to model variabilities. Other variability modeling approaches that provide variability metamodels independent from the base modeling language, are the common variability language (CVL) [22], and approaches based on aspect-oriented modeling (AOM) [12].

CVL is a general purpose variability modeling language that is designed to be woven into MOF-compliant metamodels including UML and DSLs. CVL specifies variability as a model separate from the base model and, therefore, additional mechanisms for relating elements of the variability model to elements of the base model are necessary. In AOM-based approaches (e.g., [28]) variabilities are captured as aspect models that are woven into base models, which can be described using any DSL. It is claimed that AOM can improve software modularity by localizing variability in independent aspects. Model composition or aspect model weaving are the commonly used AOM techniques to support product derivation.

Two different approaches, i.e., amalgamated and separated, to the combination of a base modeling language and a variability modeling language are identified in [22]. An amalgamated language is formed by the base

Table 3: Comparison of related work according to the modeling requirements (Section 4.3).

Related work	Base / Variability modeling notations	Variability modeling	Software modeling	Hardware and HW/SW modeling	Traceability of variabilities	Organizing and grouping var. points
Czarnecki et al. [13]	UML / annotation	Partially supported	Supported	Not addressed	Partially supported	Not supported
Santos et al. [32]	UML / UML templates	Partially supported	Supported	Not addressed	Partially supported	Supported
Gomaa et al. [21]	UML / UML profiles	Partially supported	Supported	Not addressed	Partially supported	Not supported
Ziadi et al. [38]	UML / UML profiles	Partially supported	Supported	Not addressed	Partially supported	Not addressed
Haugen et al. [22]	DSL / CVL	Partially supported	Depends on the base modeling language		Partially supported	Partially supported
Morin et al. [28]	DSL / Aspect models	Partially supported	Depends on the base modeling language		Partially supported	Not supported
Czarnecki et al. [14]	-/ EFM	Partially supported	No separated base model			Partially supported
Sinnema et al. [33]	-/ CVVL	Partially supported	No separated base model			Supported

and variability modeling languages being combined into one language (e.g., UML-based approaches such as [32, 38]), while in the separated approach the two languages are kept independent and simple references are used to relate elements from the variability model to the elements in the base model. CVL can be used in separated variability modeling approaches.

Table 3 shows our analysis of the related work mentioned above relative to the modeling requirements described in Section 4.3. Recall that these modeling requirements are particularly formulated based on the characteristics of an adequate configuration solution (Section 4.1) to address the configuration challenges in the ICS domain. To provide an analysis of related work, we have exclusively relied on what was explicitly reported in the corresponding papers. In Table 3, "Not supported" means that, according to our understanding of the work, either the variability modeling mechanism or the base modeling mechanism is unable to support the requirement; "Not addressed", on the other hand, means that the requirement is not explicitly addressed in the papers, and it is not clear whether it can be supported or not; "Partially supported" means that only some of the aspects of the requirement are fulfilled by the work, other aspects are either not supported or not addressed. In the following, we explain our analysis of the related work.

Comprehensive variability modeling. The first modeling requirement mentions that the variability modeling notation must be expressive enough to model all types of variabilities (i.e., cardinality, attribute, topology, and type variabilities). All the approaches we investigated only partially fulfill this requirement. This partial support for the variability modeling requirement is shown in the third column of Table 3. Specifically, topological variability (Section 5.6) is not addressed by any of the approaches investigated, although we can see the potential that some of these approaches (e.g., [22, 38]) can be extended to cover that case as well.

Software modeling. The second modeling requirement (the fourth column) is related to the expressiveness of the software modeling language. In the product-line modeling domain, UML is mostly used for modeling software (e.g., [13, 32, 21, 38, 15]); therefore fulfilling this requirement. Other approaches are CVL and AOM-based approaches, which model variability independently of the base model and are designed to be combined with any MOF-based language. Therefore, the ability of these approaches to support this requirement depends on the base modeling language with which they are combined.

Modeling hardware and modeling software-hardware dependencies. The third and fourth modeling requirements (the fifth column) are used to assess the ability of a product-line modeling approach in capturing hardware and its relation to software as part of the base model. None of the approaches we investigated address this need. Of course, UML-based approaches (e.g., [21, 13, 32, 38]) can be extended using, for example MARTE [1] to support hardware modeling. Also, variability modeling approaches such as CVL can be combined with DSLs (e.g., [5]) that are capable of modeling both hardware and software, for the purpose of modeling families of ICSs. However, we could not find any work providing this combination.

Traceability of variability points to elements in the base model. A variability modeling mechanism must enable tracing all the variability points to their related configurable elements in the base model. As discussed in the explanation of this requirement in Section 4.3, modeling such traceability is essential to providing a semi-automated configuration solution as formulated in Section 4. Among the approaches that we have investigated, COVAMOF (i.e., [33]) and the extended feature models (i.e., [14]) do not require traceability support, as a distinct base model does not exist in these approaches. The other approaches, either enable traceability in their variability metamodels [21, 38, 28], or provide other mechanisms for tracing variability points back to the base models [13, 32, 22]. None of these approaches, however, entirely fulfill the need for traceability in our context. This is mostly due to their inability to model certain types of variability points and their corresponding configurable elements in the base model. Therefore, as shown in column six of Table 3, we have assessed the traceability support provided by all of these approaches to be partial in our context.

Hierarchical organization and grouping of variability points. The last requirement (the last column) requires the variability modeling language to have a mechanism for explicitly grouping variability points and hierarchically organizing them into hierarchies similar to that in the base model. This is only (partially) supported by the approaches presented in [22], [32], [14], and [33].

In summary, as shown in Table 3 and discussed above, none of the investigated approaches meet all of the modeling requirements listed in Section 4.3. The main capabilities that are missing are the ability to (1) comprehensively model all types of variability points, (2) trace them back to software and hardware model elements, or (3) group them hierarchically. To fill this modeling gap, we have proposed the SimPL methodology, in which UML and its extensions are used to create both the base and the variability models. In particular, UML constructs such as classes and relationships are used to model software, four stereotypes from MARTE are used together with UML constructs to model hardware, and UML templates and packages together with three stereotypes from a newly introduced profile, named SimPL, are used to model variabilities, trace them back to the elements in software and hardware models, and organize them hierarchically according to software and hardware hierarchies.

9. Conclusion and future work

Based on a close collaboration with an industry partner and an analysis of the domain, we systematically identified and specified the configuration challenges in families of Integrated Control Systems (ICS). In such systems, large numbers of interdependent variability points and lack of adequate automation have made the configuration process a costly and error-prone task. The ultimate goal of our research is to provide an applicable configuration solution to address the configuration challenges present in the ICS domain. Such a solution is expected to improve the overall quality and productivity of the configuration process.

We argue that a configuration solution in our context should be based on concise abstractions of ICS families, and as the first step to that end, we proposed in this technical report the SimPL methodology, for creating such abstractions. Models created based on SimPL mainly target at capturing configurable components of an ICS family, their variability points, and the dependencies between them. The need for a new modeling methodology is justified to meet a set of modeling requirements derived from the characteristics of ICS families, their configuration challenges, and the characteristics of an adequate configuration solution. An analysis of the existing work in the literature shows that none of the existing approaches fulfill all of these modeling requirements. In contrast, SimPL is a methodology that is specifically designed to meet them all.

In summary, the SimPL methodology provides a notation and a set of guidelines for creating product-family models that systematically and precisely capture all the required information (i.e., commonalities and variabilities) for enabling automated configuration. To address the practical considerations of our industry partner, including training costs and availability of the modeling tools, in the design of the SimPL methodology, we relied extensively on standards: UML, an extension of it (i.e., MARTE), and OCL. In particular, we propose a UML-based variability modeling approach for modeling variability points, grouping them into reusable configuration units, and tracing them back to configurable elements in the base models (e.g., software and hardware models).

We evaluated SimPL by applying it on a large-scale industrial case study. Results of our evaluation indicate that UML, MARTE, and OCL can provide the constructs required for creating generic models of ICS families. In addition, models created by SimPL have been used as input to a prototype configuration tool presented in [8] to configure two ICS products. Results show that, using a SimPL model, the configuration tool can provide the automation described in this technical report.

In the future, we plan to conduct further empirical studies (e.g., controlled experiments and field studies) to better evaluate the SimPL methodology from different aspects such as usability and scalability.

References

[1] A UML profile for MARTE: Modeling and analysis of real-time embedded systems, May 2009.

[2] OCL: Object Constraint Language. http://www.omg.org/spec/OCL/2.2/, 2010.

[3] UML Superstructure Specification, v2.3, May 2010.

[4] Rational Software Architect V8. `http://www.ibm.com/developerworks/downloads/r/architect/`, 2011.

[5] Unified profile for DoDAF and MODAF (UPDM), Version 1.1. `http://www.omg.org/spec/UPDM/1.1/`, May 2011.

[6] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel. *Component-based product line engineering with UML*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[7] M. Becker. Towards a general model of variability in product families. In *Workshop on Software Variability Management*. 2003.

[8] R. Behjati, S. Nejati, T. Yue, A. Gotlieb, and L. C. Briand. Model-based automated and guided configuration of embedded software systems. In *ECMFA*, volume 7349 of *LNCS*, pages 226–243. Springer, 2012.

[9] R. Behjati, T. Yue, and L. C. Briand. A modeling approach to support the similarity-based reuse of configuration data. In *MoDELS*. Springer, 2012.

[10] R. Behjati, T. Yue, L. C. Briand, and B. Selic. SimPL: a product-line modeling methodology for families of integrated control systems. Technical Report Simula/TR-2011-14, 2011.

[11] J. Bosch and M. Högström. Product instantiation in software product lines: A case study. In *GCSE*, volume 2177 of *LNCS*, pages 147–162. Springer, 2000.

[12] S. Clarke and E. Baniassad. *Aspect-Oriented Analysis and Design*. Addison-Wesley Professional, 2005.

[13] K. Czarnecki and M. Antkiewicz. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, volume 3676 of *LNCS*, pages 422–437. Springer, 2005.

[14] K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Workshop on Software Factories at OOPSLA*. 2005.

[15] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE*, pages 211–220. ACM, 2006.

[16] Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005.

[17] S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *J. Syst. Softw.*, 74, January 2005.

[18] D. Dhungana, T. Neumayer, P. Grünbacher, and R. Rabiser. Supporting evolution in model-based product line engineering. In *SPLC*, pages 319–328. IEEE Computer Society, 2008.

[19] A. Egyed. Instant consistency checking for the UML. In *ICSE*, pages 381–390. ACM, 2006.

[20] H. Gomaa. *Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures*. Addison Wesley Longman Publishing Co., Inc., 2004.

[21] H. Gomaa and M. E. Shin. Automated software product line engineering and product derivation. In *HICSS*, pages 285–294. IEEE Computer Society, 2007.

[22] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, pages 139–148. IEEE Computer Society, 2008.

[23] IEEE Computer Society and ISO/IEC. IEEE-std-1471 recommended practice for architectural description of software-intensive systems. `http://www.iso-architecture.org/ieee-1471/docs/ISO-IEC-FDIS-42010.pdf`, 2011.

[24] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, 1990.

[25] K. C. Kang, S. Kim, J. Lee, K. Kim, G. J. Kim, and E. Shin. FORM: a feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5, 1998.

[26] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2), June 1992.

[27] F. J. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer-Verlag New York, Inc., 2007.

[28] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J. M. Jézéquel. Weaving variability into domain metamodels. In *MoDELS*, volume 5795 of *LNCS*, pages 690–705. Springer, 2009.

[29] R. K. Panesar-Walawege, M. Sabetzadeh, and L. C. Briand. Using UML profiles for sector-specific tailoring of safety evidence information. In *ER*, pages 362–378. Springer-Verlag, 2011.

[30] K. Pohl, G. Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., 2005.

[31] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information & Software Technology*, 52(3), March 2010.

[32] A. L. Santos, K. Koskimies, and A. Lopes. A model-driven approach to variability management in product-line engineering. *Nordic J. of Computing*, 13, September 2006.

[33] M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: a framework for modeling variability in software product families. In *SPLC*, volume 3154 of *LNCS*, pages 197–213. Springer, 2004.

[34] Matthew Stephan and Michal Antkiewicz. Ecore.fmp: A tool for editing and instantiating class models as feature models. Technical report, University of Waterloo, 200 University Avenue West Waterloo, Ontario, Canada, August 2008.

[35] M. Usman, A. Nadeem, T. Kim, and E. Cho. A survey of consistency checking techniques for UML models. In *ASEA*, pages 57–62. IEEE Computer Society, 2008.

[36] D. M. Weiss and R. Lai. *Software product-line engineering: a family-based software development process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[37] T. Yue, L. C. Briand, B. Selic, and Q. Gan. Experiences with model-based product line engineering for developing a family of integrated control systems: an industrial case study. Technical Report Simula/TR-2012-06, 2012.

[38] T. Ziadi and J. M. Jézéquel. Software product line engineering with the UML: Deriving products. *Software Product Lines*, 2006.

Appendix A. Metamodel for the modeling methodology

In this subsection, the metamodel of the SimPL modeling methodology is introduced through the class diagrams presented in Figures A.20, A.21, and A.22, along with examples. Note that the metamodel conforms to the conceptual model described in section 4.2, which discusses general configuration concepts in the domain of product-line engineering.

Appendix A.1. Viewpoint, View and Model

As discussed in the beginning of this section, the SimPL methodology conforms to the ISO/IEC/IEEE FDIS 42010 standard [23]. According to this standard, an architecture description is a collection of *architecture views* each governed by exactly one *viewpoint*. A viewpoint is a way of looking at systems; a *view* is the result of applying a viewpoint to a particular system of interest. An architecture view is itself a collection of one or more architecture models. An architecture model can be a part of more than one architecture views. Each architecture model is described using a modeling notation such as UML. These concepts and their relationships are shown in Figure A.20-(a).

The concepts of model, view, viewpoint, and their relationships are also defined in UML, which is the modeling notation that we use in the SimPL methodology. According to the UML superstructure, "A model captures a view of a system. It is an abstraction of the system, with a certain purpose." Figure A.20-(b) shows an excerpt from the UML metamodel visualizing the relationship between model and viewpoint concepts as defined in UML. As shown in this diagram, a viewpoint in UML is captured as a property of metaclass Model, indicating the name of the viewpoint that is expressed by a string.

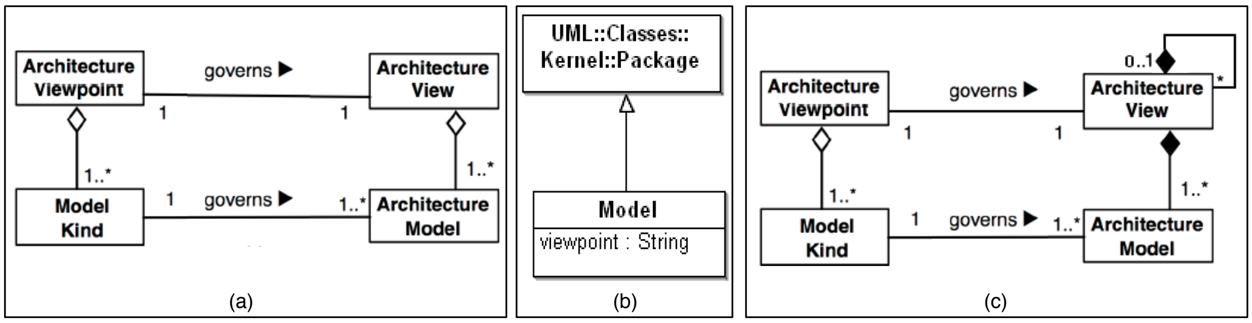


Figure A.20: A comparison between the definitions of the concepts of model, view, and viewpoint in the ISO/IEC/IEEE FDIS 42010 standard (a), UML (b), and the SimPL methodology (c).

Since UML does not provide a clear definition of view and viewpoint, and their relationships, in the SimPL methodology, we stick to the definition of these concepts from the ISO/IEC/IEEE FDIS 42010 standard but with a minor refinement. As shown in Figure A.20-(c), a self association is added to class ArchitectureView, which allows us to decompose a view into sub-views for the purpose of further separation of concerns. In addition, we have used composition associations, instead of aggregation associations to imply that a model, which is a collection of model elements, is contained by a view.

Figure A.21 shows the application of the refined ISO/IEC/IEEE FDIS 42010 standard in modeling families of ICSs as proposed by the SimPL methodology. As shown in this figure, we organize the architectural description of a family of ICSs into two views: *SystemDesignView*, which contains the base model of a product line, and *VariabilityView*, which contains the variability model of the product line. Figure A.21 also shows the decomposition of *SystemDesignView* into three sub-views: *SoftwareView*, *HardwareView*, and *AllocationView*. As depicted in Figure A.20-(c), each view and sub-view is composed of one or more architecture models. Notations and guidelines for creating such models are provided by the corresponding viewpoints. In the SimPL methodology, viewpoints governing the *SystemDesignView* and its sub-views are defined to meet requirements Req2, Req3, and Req4 described in section 4.3, while the viewpoint governing the *VariabilityView* is defined to meet Req1 and Req6.

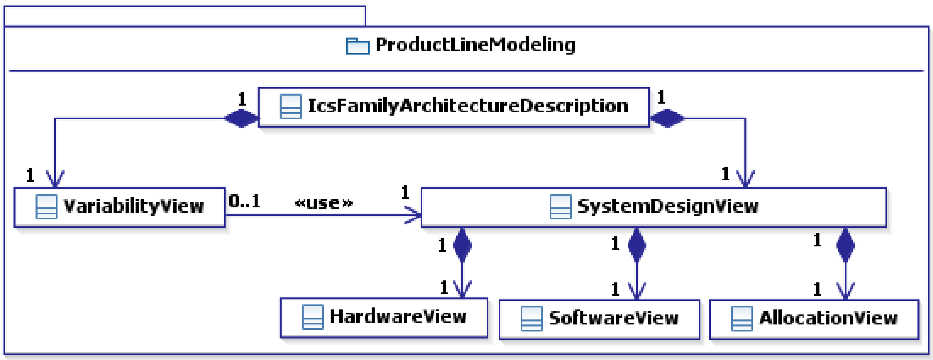


Figure A.21: Organization of the architectural description of a family of ICSs into views and sub-views.

As shown in Figure A.21, the *VariabilityView* is dependent on the *SystemDesignView*. This dependency is modeled as an association (stereotyped by «use») connecting these two metaclasses, and reflects the fact that model elements (i.e., variability points) owned by the *VariabilityView* refer to model elements (i.e., configurable elements) owned by the *SystemDesignView*. In section 5.6, we describe how these references should be captured in order to meet Req5 mentioned in section 4.3.

Table A.4 summarizes the viewpoints guiding the creation of the views and sub-views mentioned above. These viewpoints are explained in more detail in sections 5.4 and 5.6. In the SimPL profile (section 5.2), these viewpoints are implemented as a number of constrained stereotypes, which we refer to in the rest of the technical report as *viewpoint stereotypes*.

Table A.4: Summary of the view, viewpoints, and modeling notations of the SimPL methodology

Viewpoint	**Brief Description**	**Modeling notations**
Software viewpoint	Configurable software classes and the relationships among them. Most of these classes control mechanical or electrical devices.	UML classes and relationships; OCL to model constraints
Hardware viewpoint	A containment hierarchy of electrical and mechanical components and devices, and their connections. This hierarchy should include, among other elements, electrical components that have configurable software deployed on them, as well as devices that are controlled by software.	UML classes and relationships; selected MARTE stereotypes to distinguish different types of hardware components; OCL to model constraints
Allocation viewpoint	Constraints on software to hardware deployment, and interactions between software and hardware for the purpose of controlling and monitoring device operation.	«Assign» from MARTE
Variability viewpoint	Variability points, their organization into configuration units, and their references to configurable elements in the SystemDesignView.	UML templates and packages; OCL to model constraints

Appendix A.2. Views and Model Elements

Figure A.22 is an excerpt of the metamodel that shows different types of model elements contained by different models and views. As shown in this figure, views (i.e, *SystemDesingView* and *VariabilityView*) are composed of one or more model. *Model*s are composed of *ModelElements* and can either be *BaseModel* or *VariabilityModel*.

The main subtypes of *ModelElement* are *BaseModelElement*, *VariabilityModelElement*, and *Constraint*. We use *BaseModelElement* to refer to all kinds of model elements contained by the *BaseModels*. This is shown in the class diagram in Figure A.22 by the composition association connecting *BaseModel* to *BaseModelElement*.

Models in the *VariabilityView* are composed of *VariabilityModelElement*s. Two subtypes of *VariabilityModelElement* are *VariabilityPoint* and *ConfigurationUnit*. *VariabilityPoint* represents a point of variability and refers to a configurable model element (i.e., VariabilityPoint::configurableModelElement) in the *BaseModel* as the source of variability. A *ConfigurationUnit* groups the *VariabilityPoint*s that refer to a set of configurable model elements owned by a class in the *SystemDesignView* referred to as origin class (i.e., ConfigurationUnit::originClass) in the remainder of the technical report. Such an origin class represents a configurable component of the system and is associated with a *ConfigurationUnit* (i.e., Class::relatedConfigUnit). Using *ConfigurationUnit*s and their associations to origin classes fulfills Req6 mentioned in section 4.3.

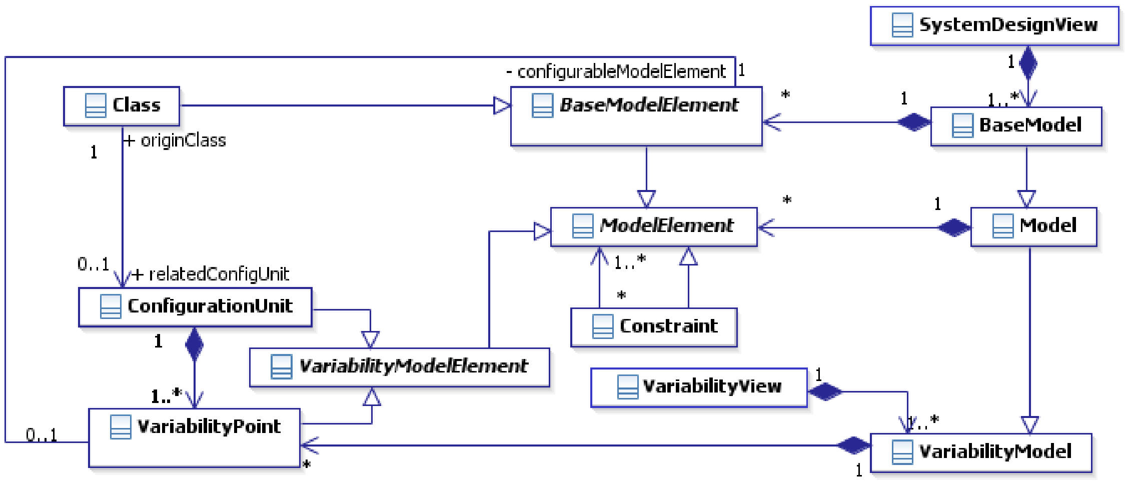


Figure A.22: An excerpt of the SimPL methodology metamodel representing the main types of model elements involved in architectural modeling of families of ICSs.

*Constraint*s are the model elements that are used, either in the *SystemDesignView* or in the *VariabilityView*, to constrain other model elements.

Appendix B. Dictionary of the SimPL profile

The SimPL profile extends UML 2 and imports four stereotypes from MARTE - namely «HwComponent», «HwComputingResource», «HwDevice», and «assign». In the SimPL profile we have introduced six constrained stereotypes that implement the viewpoints introduced by the SimPL methodology, and four more stereotypes for the purpose of distinguishing certain model elements (e.g., through «ICSystem») and supporting variability modeling (e.g., through «ConfigurationUnit», «Inherit», and «RelatedConfigUnit»).

Figure B.23 shows the dependencies of SimPL profile with OMG standards (i.e., UML, OCL, and MARTE). Specification of the 10 newly introduced stereotypes is given in the reminder of this appendix.

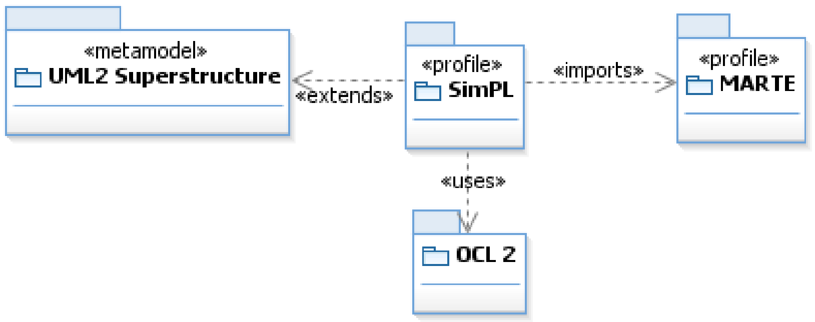


Figure B.23: Informal description of the SimPL dependencies with OMG standards.

Appendix B.1. View

«View» is an abstract stereotype implementing a viewpoint. This stereotype can be applied to a package to represent a view of an architecture description. «View» has five sub-stereotypes each representing one of the viewpoints defined in the SimPL methodology. We call these stereotypes *viewpoint stereotypes*. Figure B.24 shows the «View» and the viewpoint stereotypes generalizing it.

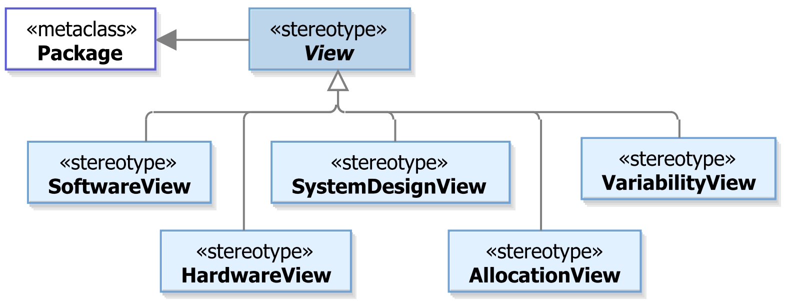


Figure B.24: Viewpoint stereotypes (constrained stereotypes applied to packages representing viewpoints).

Extensions

- Package (from UML Kernel)

Appendix B.2. SystemDesignView

The «SystemDesignView» stereotype implements the viewpoint governing the system design view. This stereotype should be applied to the package representing the system design view and containing the base models of the system.

Generalizations

- View (from SimPL)

Appendix B.3. SoftwareView

The «SoftwareView» stereotype is a subtype of «View», and implements the viewpoint governing the software sub-view. This stereotype should be applied to the package representing the software sub-view of the system. This package should be owned by a package stereotyped by «SystemDesignView».

Generalizations

- View (from SimPL)

Constraints

1. Since software sub-view is a sub-view of the system design view, a package stereotyped by «SoftwareView» should be owned by a package stereotyped by «SystemDesignView». The following OCL constraint is used to implement this.

```
context SoftwareView
self.owner.oclIsKindOf(uml::Package) and
self.owner.getAppliedStereotypes()->select(
     st: Stereotype | st.name = 'SystemDesignView')->size() = 1
```

Appendix B.4. HardwareView

The «HardwareView» stereotype is a subtype of «View», and implements the viewpoint governing the hardware sub-view. This stereotype should be applied to the package representing the hardware sub-view of the system. This package should be owned by a package stereotyped by «SystemDesignView».

Generalizations

- View (from SimPL)

Constraints

1. Similar to software sub-view, hardware sub-view is a sub-view of the system design view. Therefore, a package stereotyped by «HardwareView» should be owned by a package stereotyped by «SystemDesignView». The following OCL constraint is used to implement this.

```
context HardwareView
self.owner.oclIsKindOf(uml::Package) and
self.owner.getAppliedStereotypes()->
  select(st: Stereotype | st.name = 'SystemDesignView')->
    size() = 1
```

2. Since classes in this sub-view represent mechanical and electrical entities, they should be stereotyped by one of the MARTE stereotypes «HwComponent», «HwComputingResource», or «HwDevice». The following constraint implements this.

```
context HardwareView
self.allOwnedElements()->
  select(e:Element | e.owner.oclIsKindOf(uml::Package)
  and e.oclIsKindOf(uml::Class))->
    forAll(c : Element | c.getAppliedStereotypes()->
      select(st: Stereotype | st.name = 'HwComponent'
      or st.name = 'HwComputingResource'
      or st.name = 'HwDevice')->size() > 0)
```

Appendix B.5. AllocationView

The «AllocationView» stereotype is a subtype of «View», and implements the viewpoint governing the allocation sub-view of the system design view. This stereotype should be applied to the package representing the allocation sub-view. This package should be owned by a package stereotyped by «SystemDesignView».

Generalizations

- View (from SimPL)

Constraints

1. Since allocation sub-view is a sub-view of the system design view, a package stereotyped by «AllocationView» should be owned by a package stereotyped by «SystemDesignView». The following OCL constraint is used to implement this.

```
context AllocationView
self.owner.oclIsKindOf(uml::Package) and
self.owner.getAppliedStereotypes()->select(
     st: Stereotype | st.name = 'SystemDesignView')->size() = 1
```

Appendix B.6. VariabilityView

The «VariabilityView» stereotype is a subtype of «View», and implements the viewpoint governing the variability view. This stereotype constrains the variability models in the variability view.

Generalizations

- View (from SimPL)

Constraints

1. Since template packages in this view represent configuration units, they should be stereotyped by the «ConfigurationUnit» (section Appendix B.7) stereotype from SimPL.

```
context VariabilityView
self.allOwnedElements()->
  select(e: Element | e.oclIsKindOf(uml::Package)
  and e.allOwnedElements()->
    select(s: Element |s.owner = e
    and s.oclIsKindOf(uml::TemplateSignature))->size()=1)->
  forAll(e: Element | e.getAppliedStereotypes()->
  select(g: Stereotype | g.name = 'ConfigurationUnit')->size()=1)
```

2. Any template packages stereotyped by «ConfigurationUnit» should be referenced by a Class in the system design view through a dependency stereotyped by «RelatedConfigUnit» (section Appendix B.8).
3. A dependency between two template packages stereotyped by «ConfigurationUnit», should be stereotyped by «Inherit» (section Appendix B.9).

Appendix B.7. ConfigurationUnit

A configuration unit is a container for a number of variability points related to a configurable building block of the system. In the SimPL profile, «ConfigurationUnit» is a stereotype applicable to packages. Therefore, similar to a package a configuration unit can be used as a template. Packageable elements of such a configuration unit can be used as template parameters.

Extensions

- Package (from UML Templates)

Appendix B.8. RelatedConfigUnit

«RelatedConfigUnit» is used to distinguish a dependency connecting a model element in the system design view to its corresponding configuration unit in the variability view.

Extensions

- Dependency (from UML Kernel)

Constraints

1. «RelatedConfigUnit» can be applied only to those dependencies that connect a class in the system design view to a package stereotyped by «ConfigurationUnit» in the variability view.

Appendix B.9. Inherit

We apply «Inherit» to those dependencies connecting two configuration units. A dependency with such a stereotype applied indicates that the sub-configuration-unit inherits template parameters (indicating variability points) from the super-configuration-unit.

Extensions

- Dependency (from UML Kernel)

Constraints

1. «Inherit» can be applied only to those dependencies that connect two packages both stereotyped by «ConfigurationUnit» in the variability view. The following OCL constraint, which is owned by the «VariabilityView» stereotype implements this constraint.

```
context VariabilityView
let dependencies : Set(Dependency) =
   Dependency.allInstances()->
     select(d |  d.getAppliedStereotypes()->
       select(st | st.name = 'Inherit')->size() = 1)
in
dependencies->
    forAll(d : Dependency | d.source.getAppliedStereotypes()->
      select(st | st.name = 'ConfigurationUnit')->size() = 1
    and d.target.getAppliedStereotypes()->
      select(st | st.name = 'ConfigurationUnit')->size() = 1)
```

2. To configure an instance of the origin class of a sub-configuration-unit, it is necessary to resolve variability points listed in the sub-configuration-unit and the variability points listed in the corresponding super-configuration-unit. This constraint should be fulfilled during configuration and product derivation.

Appendix C. Specification of the architecture viewpoints

In this section, we describe the three viewpoints of the SimPL methodology using the template for documenting architecture viewpoints as described in [23].

Appendix C.1. The system design viewpoint

Overview The system design view is mainly a container for the four architecture subviews: mechanical hardware view, electrical hardware view, software view, and allocation view. The four viewpoints governing these subviews are documented in Sections Appendix C.3 to Appendix C.4 of this appendix. In this view we also capture the topmost element of the system that should be stereotyped by the «ICSystem» stereotype from the SimPL profile.

Concern The main concern in this view is to identify and capture those elements in the system that affect configuration or introduce variability. Relationships between these elements should be captured as well.

Typical stakeholders Electrical, mechanical and software engineers are the main stakeholders of the views prepared using this viewpoint. In addition, system designers can be considered as the users of the models in these views. These views, in particular, can be used as a means for better communication across different departments.

Model kind We use UML and its extensions, i.e. MARTE and the SimPL profile, to create the models in the views prepared using this viewpoint.

Correspondence rules The creation of a system design view requires creating its four subviews: mechanical hardware view, electrical hardware view, software view, and allocation view.

Creation methods Steps that should be followed to create a system design view are:

1. Create a UML package and name it SystemDesignView.
2. In the SystemDesignView package, create a UML class, C, and stereotype it by «ICSystem».
3. In the SystemDesignView create four packages representing the four subviews. Name the packages: HWView, SoftwareView, and AllocationView.
4. Proceed by creating the four subviews. These subviews contain the main subcomponents and subsystems of the system.
5. Create the relationships between class C and its subcomponents captured in the subviews of this view. We use UML relationships, associations, and dependencies for this purpose.

Appendix C.2. Software viewpoint

Overview Models in the software view capture software classes and elements that fit into at least one of the following:

- Classes that directly introduce a variability, for example through an attribute.
- Classes that are composed of other classes that introduce variability.
- Classes that has subclasses that introduce variability.

- Composition associations to create whole/part relationships between software classes.
- Generalizations to create taxonomic hierarchies of software classes.
- Associations to create relationships modeling interactions between software classes. These relationships can also introduce variability.

We use UML and a few stereotypes from the SimPL profile (e.g. «Configurable») to create models for the software view.

Concern The main concern in the software view is to identify and capture all the sources of variability in the software. It is important to provide enough information about the variability in software in order to enable supporting semi-automated software-centric configuration of families of ICSs. In addition, it is important to create software model that can easily be extended to support automatic code generation or software validation.

Typical stakeholders Software and hardware engineers creating variability models as well as configuration engineers are the main stakeholders benefiting from views created by following this viewpoint. These views, can also be used as a means for better communication across different departments.

Model kind We use UML and follow the common software modeling practices to create the models in the software view.

- We use UML classes to create software classes and entities.
- We use attributes to model attributes of a software class.
- We use the «Configurable» stereotype from SimPL to distinguish those properties and attributes that introduce variabilities.
- We use UML associations to model relationships between software classes.
- We use UML Composition associations to create whole/part relationships between software classes.
- We use UML Generalizations to create taxonomic hierarchies of software classes.

Correspondence rules

- A software view should be contained by a system design view.
- There may be relationships between elements in this view and the elements owned by the system design view. These relationships are captured as part of the system design view.
- There are dependencies and relationships between elements in this view and elements in mechanical and electrical hardware views. These relationships and dependencies should be captured as part of the allocation view.

Creation methods Steps that should be followed to create a software view are:

1. Identify the software entities representing software applications in the system, and model them as UML classes in the package named SoftwareView, which was previously created in the system design view.
2. Create composition associations between class C and the software classes created in step 1. These composition associations should be owned by system design view.
3. Proceed by decomposing each of the software classes by following the common software modeling practices using UML.

4. For each software class identify its attributes and model them using UML properties or associations. It is necessary to model all the attributes that introduce variability or affect in resolving a variability. Other attributes can be modeled as well but it is not necessary.
5. If necessary (engineers creating the software view should decide about the necessity), stereotype those attributes that introduce variability by «Configurable».

Appendix C.3. Hardware viewpoint

Overview The hardware view captures the mechanical and electrical elements of the system, including:

- Mechanical devices and instruments that are controlled by software.
- Mechanical components that physically contain devices and instruments (either mechanical or electrical).
- Mechanical components that physically contain electrical components with software deployed to them.
- Mechanical components that physically contain any of the mechanical entities listed above.
- Relationships to create containment hierarchies of the mechanical entities listed above and the electrical entities captured in the related electrical hardware view.
- Relationships to create taxonomic hierarchies for the mechanical entities listed above.
- Electrical devices and instruments that are controlled by software.
- Computing resources that have a software deployed to them.
- Electrical components that are physically composed of devices, instruments, and computing resources.
- Electrical components that are physically composed of any of the electrical entities listed above.
- Relationships to create containment hierarchies of the electrical entities listed above.
- Relationships to create taxonomic hierarchies for the electrical entities listed above.
- Relationships to model interactions between electrical entities listed above.

We use UML, MARTE and a light UML profile named SimPL to create models according to this viewpoint.

Concern The SimPL methodology is proposed to support the software-centric configuration of families of ICSs, which involve both hardware and software. The main concern for creating a hardware view in the SimPL methodology is, therefore, to provide enough information about the electrical entities involved as well as the mechanics of an ICS so that we can automatically ensure that the configured software is consistent with the hardware design. In addition, ideally, the hardware model should be used for other purposes such as functional testing and non-functional analysis of the complete system. As a result, apart from easing and supporting semi-automated configuration of ICSs, such a hardware model should be extensible in order to capture relevant details for supporting testing and analysis.

Typical stakeholders Software and hardware engineers creating variability models as well as configuration engineers are the main stakeholders benefiting from views created by following this viewpoint. These views, can also be used as a means for better communication across different departments.

Model kind We use UML and MARTE to create the models in the hardware view.

- We use UML classes stereotyped by appropriate MARTE stereotypes (e.g. «HwComponent», «HwComputingResource», and «HwDevice») to model mechanical and electrical entities.
- We use UML Composition associations to create containment hierarchies. A composition association used between mechanical or electrical entities in the views provided by this viewpoint represents physical containment.
- We use UML Generalization relationships to model taxonomic hierarchies of mechanical or electrical entities. Generalization in this context is used to factor out and encapsulate common properties and behaviors among a category of elements into a shared parent. Properties, behaviors, and relationships defined in the parent model element are reused in the child model elements. The model elements in a generalization relationship must be the same type, which in the context of the SimPL methodology includes that either the model elements have the same stereotype applied or the stereotype applied to the child be an extension of the stereotype applied to the parent model element. For example, a generalization relationship can be used between classes stereotyped by «HwComponent» or between classes stereotyped by «HwDevice»; however, it cannot be used between a class stereotyped by «HwComponent» and a class stereotyped by «HwDevice».
- We use UML associations to model interactions between mechanical entities. Such associations represent electrical links that are used for exchanging signals between electrical entities and devices.

Correspondence rules

- A hardware view should be contained by a system design view.
- There may be relationships between elements in this view and the elements owned by the system design view. These relationships are captured as part of the system design view.

Creation methods Steps that should be followed to create a hardware view are:

1. Identify the topmost hardware components of the system, and model them as classes stereotyped by «HwComponent», or «HwComputingResource» from MARTE in the package named HWView, which was previously created in the system design view.
2. Create a UML composition associations between class C and the hardware components created in step 1. These composition associations should be owned by system design view.
3. Proceed by decomposing each of the hardware components into its subcomponents and elements. UML composition associations should be used to create the hierarchy.
4. Use appropriate stereotypes from MARTE (e.g. «HwComponent», «HwComputingResource», or «HwDevice») to stereotype the newly created elements in this view.
5. Use UML Generalizations between classes in this view to create taxonomic hierarchies of mechanical entities.
6. Interactions between classes in this view should be modeled using UML associations.

Appendix C.4. Allocation viewpoint

Overview Allocation view is a mixed view in which we specify software to hardware deployment as well as software-hardware interactions. We use MARTE to model relationships in this view. Software and hardware elements are imported to models in this view from models in software and hardware views.

Concern The main concern in this view is to model the relationships and dependencies between software entities and mechanical and electrical devices. This supports keeping software and hardware views consistent. In addition, we need to explicitly model such dependencies since they may introduce variabilities.

Typical stakeholders Software and hardware engineers creating variability models as well as configuration engineers are the main stakeholders benefiting from views created by following the allocation viewpoint. These views, can also be used as a means for better communication among different departments.

Model kind We use UML and MARTE to create the models in allocation view. We use a UML comment stereotyped by «Assign» from MARTE to model software to hardware deployment. Such a comment is related to two sets of elements. The first set, named from, indicates the elements that are assigned. The other set, named to, indicated the elements to which the assignment is performed. Elements in the from list are software classes imported from models in the software sub-view, while elements in the to list are imported from models in the electrical or mechanical hardware sub-views. Elements in the to list should be stereotyped by a MARTE stereotype (e.g. «HwComputingResource»).

In addition, we use UML associations connecting a software class to a hardware resource to model that the software controls the hardware.

Correspondence rules

- An allocation view should be contained by a system design view.
- An allocation view is dependent on a software, a mechanical hardware, and an electrical hardware view.
- Models and model elements in the allocation view depend on model elements owned by the other three views of the system design view.

Creation methods Steps that should be followed to create an allocation view are:

1. Identify pairs of software classes in the software view and hardware computing resources in the electrical hardware view that introduce a software to hardware deployment relationship. Model such relationships by creating UML comments in the allocation view, stereotyped by MARTE stereotype «Assign», and each referring to a software class in their from to list, and to a hardware computing resource in their to list.
2. Identify pairs of software classes in the software view and hardware devices either in the electrical hardware view or the mechanical hardware view that introduce a software controlling hardware relationship. Model such relationships by creating associations in the allocation view, each connecting a software class to a hardware device.

Appendix C.5. Variability viewpoint

Overview The variability view is intended to provide a complete view of the system including all the variability points. We use template modeling capabilities of UML and stereotypes from SimPL profile to capture variability points. A variability point in this view refers to an element in the system design view or one of its sub-views. We organize variability points according to their origin into UML packages. In sum, the variaibility view contains:

- A number of configuration units that group variability points according to their origins.

- Each configuration unit is related to a classifier in the system design view. We use UML associations stereotyped by «RelatedConfigUnit» from the SimPL profile to model this relation. Such a UML association connects the classifier introducing variability to the package representing its related configuration unit.
- For each variability in the system we explicitly model it in this view. A variability point in this view should refer to the appropriate model element in the system design view that introduces the variability.
- Constraints and dependencies among variability points should be captured in the constraints view.

Concern The main concern in this view is to provide a complete variability model of the system that can be use to automatically configure the product-line. The references between the variability points in this view and the model elements in the system design view enables automatically creating product model from the models in the system design view of the product-line.

Typical stakeholders Members of the configuration and product derivation department are the stakeholders concerned about the views prepared using this viewpoint.

Model kind We use UML templates (Figure C.25) and SimPL profile to create the models in the variability view.

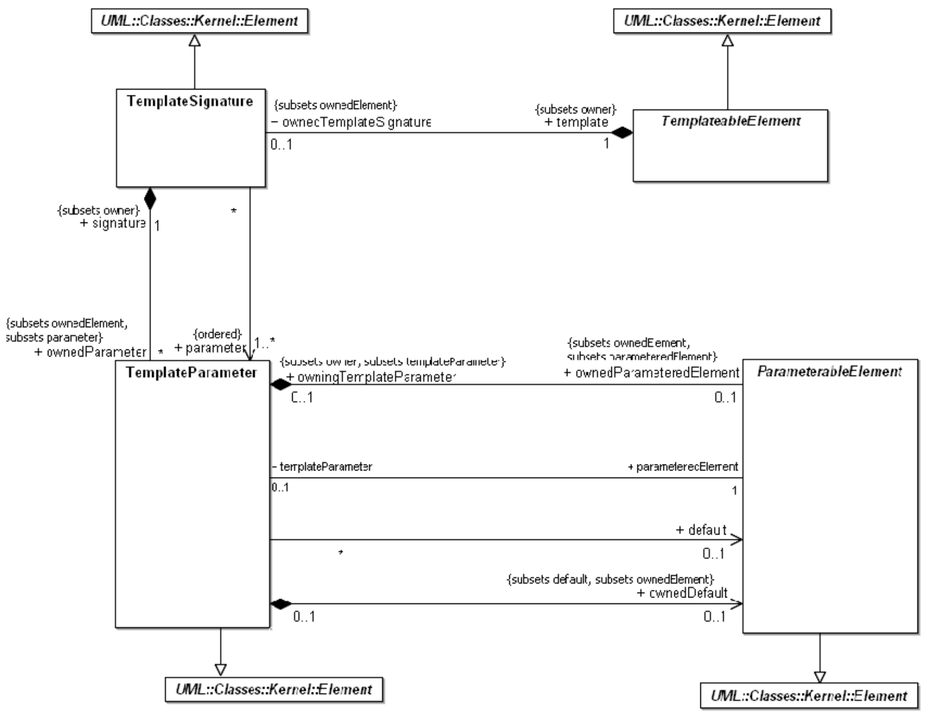

UML::Classes::Kernel::Element
UML::Classes::Kernel::Element
TemplateSignature
{subsets ownedElement}
- ownedTemplateSignature
0..1
{subsets owner}
+ template
1
TemplateableElement
{subsets owner}
+ signature
1
*
{subsets ownedElement, subsets parameter}
+ ownedParameter
*
{ordered}
+ parameter
1..*
TemplateParameter
{subsets owner, subsets templateParameter}
+ owningTemplateParameter
0..1
{subsets ownedElement, subsets parameteredElement}
+ ownedParameteredElement
0..1
ParameterableElement
- templateParameter
0..1
+ parameteredElement
1
+ default
0..1
*
{subsets default, subsets ownedElement}
+ ownedDefault
0..1
0..1
UML::Classes::Kernel::Element
UML::Classes::Kernel::Element

Figure C.25: UML templates (Figure 17.16 of [3]).

Figure C.25 shows elements in the Templates packages of the UML metamodel that we use to create the variability models in the variability view. These elements include TemplateableElement, TemplateParameter, and ParameterableElement. A template is a parameterized model element that describes or identifies the pattern for a group of model elements of a particular type. A templateable element is an element (e.g., Class, Package) that can optionally be defined as a template; a template parameter exposes a parameterable element as a formal template parameter of a template; a parameterable element is an element that can be exposed as a formal template parameter for a template.

- We use UML packages stereotyped by «ConfigurationUnit» from the SimPL profile to model configuration units.
- We use UML associations stereotyped by «RelatedConfigUnit» from the SimPL profile to associate a configuration unit to its corresponding classifier in the system design view.
- The classifier that introduces variability and its owned elements should be imported to the variability view.
- For each variability in the system we explicitly model it in the variability view using a UML *template parameter* in the appropriate package (the package that is related to the classifier that introduces the variability). A variability point in this view should refer to the appropriate model element in the system design view. Such a model element represents a *parametrable element* and should be owned by the classifier that is related to the package that owns the template parameter.

Correspondence rules

- The variability view is dependent on the system design view.
- Each package in the variability view is related to a classifier in one of the sub-views of the system design view. These relationships are captured as part of the variability view.

Creation methods To make the variability view we first create a package and name it VariabilityView. Then, we start from the topmost class captured in the system design view, stereotyped by «ICSystem» and perform the following steps:

1. For each configurable class e_i in the system design view, we create a package cu_i in the package VariabilityView and stereotype it by «ConfigurationUnit».
2. We add a dependency between e_i and cu_i, and stereotype it by «RelatedConfigUnit».
3. For each variable model element (e.g. an attribute), v_{ij}, stereotyped by «Configurable» and contained by e_i, we create a TemplateParameter, t_{ij}, in cu_i referring to v_{ij}.

Notes The packages stereotyped by «ConfigurationUnit» are TemplateableElements and each have a TemplateSignature that contains all the template parameters (t_{ij}).

Relationships that exist between variable classes affect the design of the variability view. Consider two variable classes e_p and e_q. If e_q is a subclass of e_p then cu_q should extend cu_p, which means that cu_q inherits all the template parameters from the signature of cu_p. In order to do this, we add a dependency connecting cu_q to cu_p and stereotype it by «Inherit». In addition, a composition relation connecting e_p to e_q implies that there is a nesting of variabilities between cu_p and cu_q. We do not explicitly model this nesting, since it can be derived from the composition relationship between e_p and e_q and the dependencies that, respectively, connect e_p and e_q to cu_p and cu_q.

Appendix D. OCL constraints

In this section, we present the OCL constraints that we defined as part of the case study reported in this technical report.

Appendix D.1. Domain-specific constraints

In this section, we describe the domain specific constraints, i.e., application independent constraints. These are the constraints that should be modeled using OCL as part of the product-line model. We need to use OCL, since UML class and relationship concepts are not sufficient for expressing such constraints.

The following constraint is written in the context of the class ElectricalConnection, and makes sure that, the value of the pin associated with a device is in the valid range.

```
context ElectronicConnection inv PinRange
pinIndex >= 0 and sem.eBoards->asSequence()->
     at(ebIndex+1).numOfPins > pinIndex
```

The OCL constraint BoardIndRange below, expresses that the value of the attribute ebIndex should be within the valid range, i.e., withing the list of indexes of the electronic boards of the respective SEM.

```
context ElectronicConnection inv BoardIndRange
ebIndex >= 0 and ebIndex < sem.eBoards->size()
```

The following constraint makes sure that no two devices are connected to the same pin of the same board. In other words, if two devices are located on the same electronic board, they should be connected to different pins.

```
context SEM inv PinIndUnique
connections->isUnique(c : Connection | c.eBoardIndex * 100 + c.pinIndex)
```

Another constraint on SEM is:

```
context SEM inv DeviceAndControllerMap
let
sensors : Set(Sensor) =
            Sensor.allInstances()->select(s | self.devices->includes(s)),
valves : Set(Valve) =
            Valve.allInstances()->select(v | self.devices->includes(v)),
chokes: Set(Choke) =
            Choke.allInstances()->select(c | self.devices->includes(c)),
sControllers : Set(SensorController) = SensorController.allInstances()->
            select(sC| self.deployedSemApp.controllers->includes(sC)),
vControllers : Set(ValveController) = ValveController.allInstances()->
            select(vC | self.deployedSemApp.controllers->includes(vC)),

cControllers : Set(ChokeController) = ChokeController.allInstances()->
            select(cC | self.deployedSemApp.controllers->includes(cC))
in
sControllers.sensor->includesAll(sensors)
and
vControllers.valve->includesAll(valves)
and
cControllers.choke->includesAll(chokes)
```

Another group of constraints are specified to make sure that a device controller controlling a device contains correct information about the location (i.e., electrical board and pin indexes) of that device. The following is the constraint that we wrote for the SensorController.

```
context SensorController inv SensorPinCompatibility
let
dcs : Set(Connection) = Connection.allInstances()->
           select(dc | dc.device = self.sensor),
dc : Connection = dcs->asOrderedSet()->at(1)
in
dcs->size() = 1 and
dc.eBoardIndex = self.eBoardIndex and
dc.pinIndex = self.pinIndex
```

Two more constraints similar to the one above are defined for ChokeController and ValveController.

Another constraint on Valve:

```
context ValveControllers inv Valve
let
openVCtrls : Set(ValveController) = ValveController.allInstances()->
            select(v: ValveController | v.valve = self.choke.openValve),
closeVCtrls : Set(ValveController) = ValveController.allInstances()->
            select(v: ValveController | v.valve = self.choke.closeValve)
in
openVCtrls->includes(openValveController)
and
closeVCtrls->includes(closeValveController)
```

A constraint on TemplatedSubseaField:

```
context TemplatedSubseaField inv SEMsSimilarity
templates.xmasTree.SCM.sems->
      union(templates.manifold.MCM.sems)->forAll(s, t | s.eBoards = t.eBoards)
```

This constraint specifies that all SEMs in a subsea field should have the same number and types of eBoards. A similar constraint is defined for scattered subsea field:

```
context ScatteredSubseaField inv SEMsSimilarity
manifold.MCM.sems->
      union(xmasTrees.SCM.sems)->forAll(s, t |s.eBoards = t.eBoards)
```

Note that in the two constraints above, equation of two eBoards implies that the lengths of the two sets are equal and moreover they contain the same elements.

Four constraints are defined to ensure the relationship between devices and the software controlling them. DeviceControlling1 is defined on the DeviceControl association class between SEM and Device

```
context DeviceControl inv DeviceControlling1
let
controllers : Set(DeviceController) = sem.deployedSemApp.controllers
in
sem.deployedSemApp->size() = 1
and (
controllers->exists(c| c.oclIsTypeOf(SensorController) and
c.oclAsType(SensorController).sensor = device)
or
controllers->exists(c| c.oclIsTypeOf(ChokeController) and
c.oclAsType(ChokeController).choke = device)
or
controllers->exists(c| c.oclIsTypeOf(ValveController) and
c.oclAsType(ValveController).valve = device)
)
```

DeviceControlling2 describes that devices controlled by the SEMs of a Xmas tree are a subset of the devices owned by that Xmas tree. This constraint together with DeviceControlling1 on the controls relationship between SEM and Device, ensures that all the devices controlled by the software deployed to the SEMs of a Xmas tree are a subset of the devices owned by that Xmas tree.

```
context XmasTree inv DeviceControlling2
devices->includesAll(SCM.sems.devices)
```

DeviceControlling3 specifies that all the devices controlled by the SEMs of a Manifold either belong to the Manifold or belong to one of its related Xmas trees.

```
context Manifold inv DeviceControlling3
device->union(relatedXT.devices)->
     includesAll(MCM.sems.devices)
```

DeviceControlling4 specifies that all the device in each Xmas tree are either controlled by at least one of the SEMs of that Xmas tree or by at least one of the SEMs of its related Manifold.

```
context XmasTree inv DeviceControlling4
SCM.sems.devices->
     union(relatedManifold.MCM.sems.devices)->asSet()->includesAll(devices)
```

Appendix D.2. Application-specific constraints

These are the constraints that only apply to a specific applications. This group of constraints are also specified in OCL and are provided by the user separately for each application. A group of these constraints can be seen as configuration rules that restrict and guide configuration.

For example we can consider *redundancy* as a feature of the system. If this feature is selected, it means that:

- Each subsea control module in the system has two subsea electronic modules. This can be specified as a simple OCL constraint on the class ControlModule: self.sems->size() = 2 (i.e., constraint OptFtrRedundancy in ControlModule).
- The software deployed to both subsea electronic modules should have the same structure. It means that both sem applications should have the same number of instances of each subclass of DeviceController. Also, since those instances of DeviceController in semA and instances of DeviceController in semB control the same set of devices, they share some of their configuration. However, this shared configuration can be derived from the product-line model, including the classes, their relationships, and the additional OCL constraints specified on the model. (add a constraint to product-line: for each device in a Xmas tree, the semApp deployed to any of the sems of the control module of that Xmas tree should have a devicecontroller connected to that device.)

To distinguish OCL constraints that represent feature from other constraints that apply to all members of the family, we use a naming convention: the constraints representing features start with "OptFtr". This naming suggests that the constraint represents an optional feature. If the feature is selected during the configuration of a specific product, the constraint should be preserved during the rest of the configuration process.

In addition to the redundancy feature, three more constraints are defined in the model. These constraints are:

This constraint ensures that all subsea fields in a system are the same and all are templated.

```
context FMCSystem inv OptFtrAllTemplated
subseaFields->forAll(oclIsTypeOf(TemplatedSubseaField))
```

This constraint ensures that all subsea fields in a system are the same and all are scattered.

```
context FMCSystem inv OptFtrAllScattered
subseaFields->forAll(oclIsTypeOf(ScatteredSubseaField))
```

Note that, there is an XOR relationship between OptFtrAllTemplated and OptFtrAllScattered. We can use a simple feature model to show this. We can define certain stereotypes to distinguish between such optional features and other constraints, since the constraints representing features do not represent mandatory invariants and sometimes they are inconsistent.

The last product specific constraint specifies that all the templates in a templated subsea field should have similar configurations. To what degree the configurations should be similar is specified in the constraint. We can define more constraints and associate them with some new features to provide higher levels of similarity.

```
context TemplatedSubseaField inv OptFtrTemplateSimilarity
templates->forAll(t1, t2: Template | t1.xmasTree->size() = t2.xmasTree->size()
and
t1.xmasTree.SCM->select(oclAsType(SCM).type = CMType::INJECTION)->size() =
   t2.xmasTree.SCM->select(oclAsType(SCM).type = CMType::INJECTION)->size()
and
t1.xmasTree.SCM->select(oclAsType(SCM).type = CMType::PRODUCTION)->size() =
   t2.xmasTree.SCM->select(oclAsType(SCM).type = CMType::PRODUCTION)->size()
and
t1.manifold.MCM.sems->size() = t2.manifold.MCM.sems->size()
and
t1.manifold.device->select(oclIsTypeOf(Sensor))->size() =
    t2.manifold.device->select(oclIsTypeOf(Sensor))->size()
and
t1.manifold.device->select(oclIsTypeOf(Choke))->size() =
    t2.manifold.device->select(oclIsTypeOf(Choke))->size()
and
t1.manifold.device->select(oclIsTypeOf(Valve))->size() =
     t2.manifold.device->select(oclIsTypeOf(Valve))->size()
)
```