A Model-Based Approach to the Software Configuration of Integrated Control Systems

by
Razieh Behjati

Thesis submitted for the degree of Philosophiae Doctor
Department of Informatics
Faculty of Mathematics and Natural Sciences
University of Oslo
August 2012

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1246

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Akademika publishing.
The thesis is produced by Akademika publishing merely in connection with the thesis defence. Kindly direct all inquiries regarding the thesis to the copyright holder or the unit which grants the doctorate.

Abstract

Software product-line engineering is a paradigm for developing software applications through reuse and mass customization. A product family provides a repository of reusable components, where each component has a number of configurable features. Product development in this context is done through configuration, which is the process of selecting and customizing the reusable components according to the specific needs of a particular product. Software product-lining has been extensively applied in the design and development of integrated control systems, which are large-scale, heterogeneous, and hierarchical systems typically used in the oil and gas domain. Due to the complexity of such systems, the lack of concise abstractions, and inadequate automation support, product configuration, in the integrated control systems domain, is typically error-prone and laborious.

In this thesis, we identify and formulate the configuration challenges in the integrated control systems domain, and propose a model-based semi-automated configuration approach to overcome those challenges. Our solution to the configuration problems consists of a UML-based modeling methodology, named SimPL, and a semi-automated configuration approach. The SimPL methodology enables creating concise abstractions of families of integrated control systems. Our semi-automated configuration approach uses constraint satisfaction techniques to provide automation support for deriving products that are guaranteed to be consistent with the SimPL models of their respective product families.

We have performed a comprehensive domain analysis to identify characteristics of families of integrated control systems, and their configuration challenges. We then derived a set of modeling requirements based on the findings of our domain analysis. The SimPL methodology is proposed to fulfill these requirements. We have defined and formalized the notion of product configuration and its consistency, in the integrated control systems domain, and we have provided mathematical analysis to prove that our approach to configuration ensures the consistency and the correctness of the derived products with respect to their product family models. We have implemented our configuration approach in a configuration engine and we have evaluated its capabilities by applying it to a family of real subsea oil production systems from our industry partner.

To evaluate the ability of the SimPL methodology in fulfilling the modeling requirements, we applied it to a large-scale industrial case study. Our experience with the case study shows that the SimPL methodology can provide a model of the product family that meets all the modeling requirements. Moreover, our experiments with the configuration engine shows that up to 50% of the configuration decisions can be automated using our approach, therefore reducing configuration effort. Furthermore, by taking into account the internal similarities, our approach can offer a higher automation rate of more than 60%.

In conclusion, the research presented in this thesis shows that software configuration in the domain of integrated control systems can be mechanized and automated to a considerable extent. Such automation support can reduce configuration effort and configuration complexity, and can ensure the consistency of final products. Moreover, our work shows that UML-based modeling methodologies, such as SimPL, can be tailored to provide the foundation required for providing the automation support.

Aknowledgements

First and foremost I would like to thank my supervisors Lionel Briand, Shiva Nejati, and Tao Yue. Their insightful guidance and support has been invaluable. Their scientific and practical advise have particularly helped me develop my analytical skills and improve my scientific presentation skills. I have learned from them how to do research and I am grateful for their high standards for work, kindness, friendliness, and sincerity.

I am grateful to Bran Selic and Arnaud Gotlieb for all their thoughtful guidance and scientific support at various stages during my PhD.

Special thanks to Simula Research Laboratory and Simula School of Research and Innovation for providing an excellent work place. I should also thank my colleagues at Simula. It was both inspiring and fun to interact with them and learn from them with all their different academic and cultural backgrounds.

I would also like to thank the people at our industry partner, FMC Technologies. Collaboration with them, and their help gave me the opportunity to understand the needs of industry and work with industrial case studies - both fundamental in completing this thesis.

Last but not least, I would like to thank my family and friends. I am mostly grateful to my wonderful parents for all their love and support throughout these years.

Razieh Behjati, August 2012

Contents

List of Papers vii

Summary **1**
1 Introduction 1
2 Background 5
3 Model-Based Configuration 11
4 Research Method 17
5 Summary of Results 21
6 Directions for Future Work 26
7 Conclusion 27

Paper 1: SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems **31**
1 Introduction 34
2 Motivation and scope 36
3 ICS families: characteristics and configuration challenges 38
4 Solution overview 44
5 The SimPL modeling methodology 51
6 Product configuration 77
7 Evaluation and discussion 79
8 Related work 86
9 Conclusion and future work 90

Paper 2: Architecture-Level Configuration of Large-Scale Embedded Software Systems: A Formal Specification **95**
1 Introduction 98
2 Configuration of ICSs: Practice and Problem Definition 99
3 The SimPL methodology 101
4 Formal specifications 106
5 The configuration process 117
6 Semi-automated configuration 122
7 Characteristics of the semi-automated configuration 132
8 Related work 135
9 Conclusion and future work 137

Paper 3: Model-Based Automated and Guided Configuration of Embedded Software Systems **143**

1 Introduction . . . 146
2 Configuration of ICSs: Practice and Problem Definition . . . 147
3 Overview of our approach . . . 149
4 Product-line modeling . . . 150
5 Interactive model-based guided configuration . . . 153
6 Prototype tool . . . 156
7 Evaluation . . . 159
8 Related Work . . . 164
9 Conclusion . . . 165

Paper 4: A Modeling Approach to Support the Similarity-Based Reuse of Configuration Data **169**
1 Introduction . . . 172
2 Configuration reuse: practice and problem definition . . . 173
3 Related work . . . 175
4 Overview of our approach . . . 176
5 A subsea product-family model . . . 179
6 Similarity modeling . . . 181
7 Similarity configuration . . . 186
8 Configuration reuse through constraint propagation . . . 187
9 Evaluation . . . 188
10 Conclusion . . . 190

List of Papers

Paper 1. SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems

Razieh Behjati, Tao Yue, Lionel Briand, and Bran Selic
Accepted for publication in the Journal of Information and Software Technology, 2012

Paper 2. Architecture-Level Configuration of Large-Scale Embedded Software Systems: A Formal Specification

Razieh Behjati, Shiva Nejati, and Lionel Briand
Submitted to ACM Transactions on Software Engineering and Methodology, 2012.

Paper 3. Model-Based Automated and Guided Configuration of Embedded Software Systems

Razieh Behjati, Shiva Nejati, Tao Yue, Arnaud Gotlieb, and Lionel Briand
Published in the proceedings of the 8th European Conference on Modeling Foundations and Applications, ECMFA 2012.

Paper 4. A Modeling Approach to Support the Similarity-Based Reuse of Configuration Data

Razieh Behjati, Tao Yue, and Lionel Briand
Published in the proceedings of ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems, MODELS 2012.

The four papers listed above are self-contained. Therefore, some information is repeated. There are also some differences in the terminologies used in the papers.

My contributions
For all papers, I was responsible for the idea, implementation, experiments design, analysis, and writing. My supervisors contributed in all phases of the work.

During my PhD study, I also contributed another paper that is not included in this thesis.

Extending SysML with AADL Concepts for Comprehensive System Architecture Modeling

Razieh Behjati, Tao Yue, Shiva Nejati, Lionel Briand, and Bran Selic
Published in the proceedings of the 7th European Conference on Modeling Foundations and Applications, ECMFA 2011.

Summary

1 Introduction

Modern society is increasingly dependent on integrated or embedded control systems. Examples of such systems include oil and gas production platforms, industrial robots, and automotive and avionics systems. Integrated control systems are heterogeneous systems that combine mechanical, electrical, and software components. They are large-scale both with respect to the diversity of the types of their contained hardware and software components (i.e., tens of component types) and the number of components that a system typically contains (i.e., thousands of hardware and software component instances). These systems are usually hierarchical, with complex components containing other finer-grained components.

The heterogeneous nature of integrated control systems, their scale, and the complexity in their functionality, have made the production of such systems laborious and costly. To improve quality and to reduce the overall engineering effort and production costs, many organizations has turned towards various reuse strategies. In particular, many organizations have adopted software product-line engineering approaches to develop the software embedded in their systems. These product lines 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, involves selecting and customizing (through assigning values to configurable parameters) the reusable components according to the specific needs of a particular product. We refer to this as the *configuration* process.

Configuration of software in the integrated control systems domain is complicated by a number of factors. These factors are largely due to the complexity of these systems and ineffective adoption of product line engineering approaches. The latter can be characterized by

its support for *abstraction* and *automation* [16]. Abstraction, in general, plays a central role in software reuse. Concise and expressive abstractions are required to effectively specify related collections of reusable artifacts. Automation, on the other hand, is required for effective and error-free selection and customization of reusable artifacts. 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 based on concise abstractions of the reusable artifacts becomes crucial to the configuration process [7, 19, 20]. In practice, however, many adoptions of product line engineering lack a concise and communicable abstraction of their reusable artifacts, and define configuration processes that involve manually selecting components and manually assigning values to tens of thousands of configurable parameters [7, 10, 17].

Inadequate (or lack of) automation for software configuration and the complexity of integrated control systems – which require the manual configuration of a large number of interdependent configurable parameters – result in increased opportunity for configuration errors. Most of these configuration errors are revealed very late, during integration testing, when the configured software and hardware are integrated. Localizing errors and fixing them at this stage is very costly. In many cases, configuration errors are mistakenly reported as software errors or integration errors (e.g., interface mismatch between hardware and software), making the debugging process even more expensive and lengthy.

This thesis provides a coherent configuration solution to overcome the configuration challenges in the integrated control systems domain. Our objective is to reduce the costs of software configuration while improving the quality of the configured software. We propose a model-based configuration approach that (1) detects configuration errors early during the configuration process by iteratively validating configuration decisions, (2) reduces the complexity of making consistent configuration decisions by interactively guiding configuration engineers throughout the configuration process and (3) reduces the configuration effort by automatically making some of the configuration decisions. The basis of our configuration approach is a modeling methodology, named SimPL, which is devised based on industry standards (i.e., UML and its extensions). The SimPL methodology provides notation and guidelines for creating concise abstractions of reusable artifacts in product lines of integrated

control systems. To validate configuration decisions, provide user guidance, and automate configuration decisions, we use constraint solving over finite-domains [9]. To further reduce configuration effort and enhance the practical adoption of our solution, we have proposed a reuse-oriented configuration approach that enables automated reuse of configuration decisions based on the internal similarities that exist within individual products.

Contributions

This thesis focuses on the problems faced by organizations, in the integrated control systems domain, when adopting product line engineering for producing the software embedded in their systems. Contributions of this thesis are related to both the abstraction and the automation required for effective product line engineering. In particular:

1. We have identified the essential characteristics of a configuration solution in the integrated control systems domain. These characteristics are derived from our collaboration with industry partners and similar experiences reported in the literature. Based on these characteristics, we have derived a set of modeling requirements for creating concise and expressive abstractions of families of integrated control systems.

2. We have designed and developed a modeling methodology (named SimPL) that fulfills the modeling requirements. This modeling methodology is based on industry standards (i.e., UML and its extensions) and provides a notation and a set of guidelines for creating models of product lines in integrated control systems domain.

3. We have designed and implemented an iterative and interactive semi-automated configuration approach that enables consistent and error-free configuration of software through (1) automatically evaluating each configuration decision (i.e., the value to be assigned to a configurable parameter), (2) interactively guiding the configuration engineers (i.e., the individuals who do the configuration) during the configuration process, and (3) automatically inferring some of the configuration decisions.

4. We have proposed a reuse-oriented configuration approach to enable the automated reuse of configuration data based on the internal similarities in a single product. The

reuse-oriented configuration approach is an extension of our semi-automated configuration approach. It consists of a similarity modeling approach, devised based on the SimPL methodology, that allows a higher degree of reuse. Similarity-based reuse of configuration data is shown to be effective, especially, for embedded software systems with a high degree of internal similarities (i.e., structural similarities across various components of an individual system).

5. We have applied our approach on a product family from our industry partner. Three large-scale products have been studied and used in different steps of this thesis for evaluating the modeling methodology, the semi-automated configuration approach, and its reuse-oriented extension. Industrial case studies of this size are rarely reported in the literature.

6. We have formalized the notion of consistent configuration. As part of this formalization, we have defined mathematical structures for describing product families (including reusable components and their configurable parameters) and products. Using the mathematical structures we have redefined the configuration problem as a constraint satisfaction problem. The configuration process is, thereupon, redefined in terms of the constraint solving concepts and operations that allows us prove that our configuration approach can ensure the consistency of the derived products.

Thesis Structure

This thesis is a collection of papers and is organized into two parts:

Summary: This part summarizes the research conducted for this thesis and introduces the included papers. In Section 2, background information on the main concepts discussed in this thesis are presented. In Section 3, the core ideas of the thesis are explained. Section 4 explains the research method employed and Section 5 provides a summary of the main results. Section 6 discusses the future direction for this research and Section 7 concludes.

Papers: The rest of the thesis consists of four published, accepted for publication, and submitted papers in international journals and peer-reviewed conferences. Paper1 covers the

first two contributions mentioned above. Paper3 and Paper4 cover the third and the fourth contributions, respectively. The last contribution is covered in Paper2. An overview of these papers is presented in Section 5 of this summary.

2 Background

In this thesis we propose a model-based solution to the challenges facing the configuration of software in the integrated control systems domain. In this section, we provide the background information on the main concepts involved in this thesis. First we give an overview of model-based software engineering, including a brief explanation of the modeling standards that we use. Then we provide a brief introduction to the idea of software reuse through product-line engineering.

2.1 Model-Based Software Engineering

Models have been used in all traditional engineering disciplines as the basis for understanding complex problems and their potential solutions. In our context, a model is an abstraction of a system, which retains only the information that is relevant for a specific purpose. Due to the increasing complexity of software systems, models that provide concise representations of systems at various levels of abstraction become vital for a software engineering approach to succeed.

Model-based software engineering (MBSE) is a discipline where models are created and used as a basis for understanding a domain, and developing (e.g., designing, implementing and evaluating) a software solution. A model, in this context, represents a software artifact or a real-world domain and should conform to a metamodel [12]. Metamodels provide a means for defining modeling languages. A metamodel provides a set of constructs and rules needed to build specific models within a particular domain of interest.

In this thesis, we rely on industry standards for creating the models required for product development through configuration. Industry standard modeling languages are more likely to be known by the people in industry. This makes it easier for our configuration solution to be

adopted by industrial organizations. Moreover, relying on standards provides us with a wide range of related technologies and tools which are central if the approach is to be employed in practice. In particular, we use UML, its extensions, and OCL to create models of product lines. In the remainder of this section UML, OCL, and the extension mechanisms of UML are briefly introduced.

2.1.1 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is a de facto modeling standard created and managed by the object management group (OMG). UML is a general-purpose modeling language that provides a rich set of modeling constructs and notations for modeling both structural and behavioral aspects of a system with a special focus on software modeling. UML constructs enable object-oriented design and generic modeling.

We use UML to create generic models of product families. A generic model specifies a group of similar products or systems, i.e., a *product family*. The key techniques for creating generic specifications are parameterization, information hiding, and inheritance [13]. UML equips us with these techniques through its structural modeling constructs. In particular, we use classes, properties, and relationships (e.g., associations and generalization) to create models of integrated control systems families. In addition, we use UML template modeling constructs to explicitly capture the reusable components of a system.

2.1.2 Object Constraint Language (OCL)

The Object Constraint Language (OCL) [4] is a declarative language for writing constraints on UML models. OCL is based on first-order predicate logic but it uses a syntax similar to programming languages. The language also provides a standard library that defines a number of operations on various OCL types (e.g., collections). OCL can be used for a number of different purposes. These include defining invariants on classes, writing pre- and post-conditions on operations, and querying a model. OCL invariants are used to express additional constraints on the instances of a class that cannot be expressed, or are very difficult to express, with the graphical means provided by UML. An OCL invariant written in the context of a class should be true for all instances of that class.

In our approach to configuration, OCL invariants on software and hardware classes play a central role. OCL constraints, in a product family model, implement the consistency rules in the domain, define additional restrictions on the relationships between reusable classes and components, and express the dependencies between their configurable parameters. OCL constraints defined as part of a product family model provide the information required for validating user-provided configuration decisions, inferring configuration decisions, and providing user guidance.

2.1.3 UML Extension Mechanisms (Profiles)

UML can be extended, for example by introducing new model elements, to meet the needs of a special domain. Profiles are the UML mechanism for extending the language. In a profile, the basic UML constructs are customized and extended with new semantics by using four UML extension mechanisms defined in the UML specification [2]: stereotypes, tag definitions, tagged values, and constraints.

Using stereotypes one can define new model elements that assign additional semantics to the basic elements in UML. Tag definitions can be attached to model elements. They allow one to introduce new kinds of properties that the model elements may have. The value assigned to a tag definition is a tagged value. Constraints can be used to further refine the semantics of the model elements. In a profile, constraints are usually defined using OCL expressions attached to some stereotypes. More details on UML extension mechanisms can be found in [2].

Many important UML profiles have now been developed, and some of them are adopted and standardized by OMG. Two examples of these profiles are SysML [6, 11] and MARTE [3]. SysML, the OMG System Modeling Language, is a general-purpose modeling language for systems engineering and extends a subset of UML metamodel. MARTE is the UML profile for Modeling and Analysis of Real-Time and Embedded Systems.

As part of the work in this thesis, we have developed a UML profile, named SimPL, that facilitates creating generic models of integrated control systems families. In this profile, we have imported several stereotypes from MARTE to enable hardware modeling. Additional stereotypes are defined to enrich the models with information required for automated config-

uration. Several OCL constraints are defined in the SimPL profile. These OCL constraints implement a set of consistency rules that should be preserved in the product family models. Consistency of the product family models can be ensured, to a certain extent, using these constraints. Details about the SimPL profile is presented in Paper1.

2.2 Product-Line Engineering

Software product line engineering [18] is a paradigm for developing software applications through reuse and mass customization. Its objective is to improve quality and to reduce the overall engineering effort and development cost by broadening the traditional software development approaches to consider a *product family* instead of focusing on a single software system [13]. A product family is a collection of similar software systems that have some common functionality, but vary in some aspects or features. To take advantage of the common functionality, reusable artefacts (e.g., architecture, design, components) are developed, which can be customized and reused by different members of the family.

Commonly, a software product line engineering framework distinguishes two processes: the *domain engineering* and the *application engineering* processes [18]. Domain engineering focuses on a product family as a whole. During this process commonalities and variabilities are defined and the reusable artifacts are developed. Application engineering, on the other hand, focuses on the production of a particular product from the product family assets and artifacts. One major step during production is the configuration process[1]: the set of activities required for the selection and customization of reusable components according to the needs of a product.

Figure 1 shows the two processes and the basic sub-processes in each process. In practice, variations of these processes are adopted by companies delivering product lines. For example, in the embedded systems domain new sub-processes may be needed to deal with hardware development, or to cope with legacy systems some sub-processes may need to be done differently. In the remainder of this section we only explain the basic activities that are performed in each sub-process.

[1]The configuration process is in fact an stage of the application design sub-process (Figure 1).

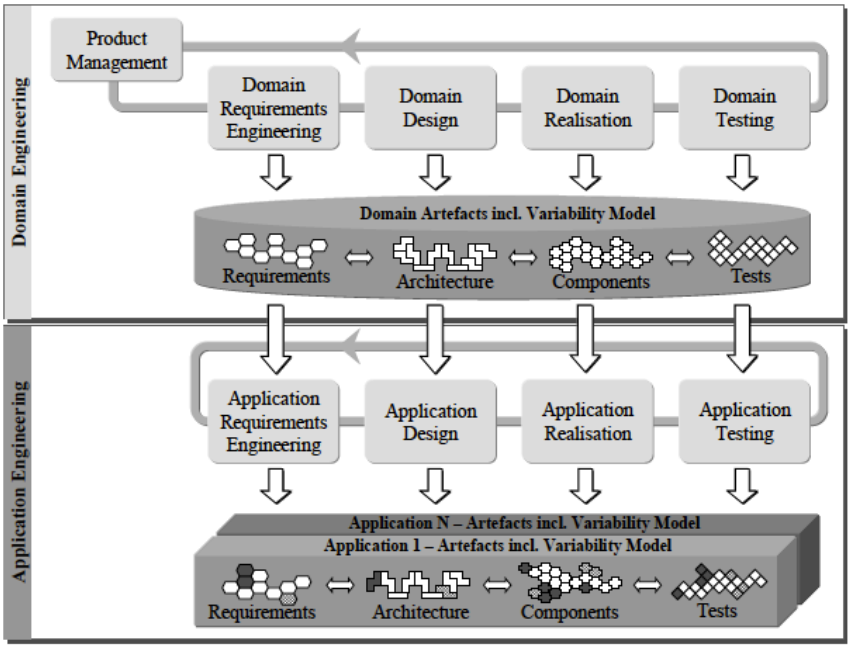


Figure 1: The software product-line engineering framework (from [18]).

Domain engineering sub-processes

Domain engineering is usually an incremental process. The main sub-processes that should be performed during domain engineering are listed below:

- **Product management.** During the product management sub-process, economic aspects of the software product line are studied to provide a product roadmap that determines the major common and variable features.
- **Domain requirements engineering.** During this sub-process common and variable requirements for the product family are elicited and documented. Common requirements specify the main functionality of the products in the product family. Variable requirements specify optional functionality or quality attributes that some products may possess. Output of this sub-process comprises reusable, textual and model-based requirements. To cope with the evolution of the product family, domain requirements engineering anticipates prospective changes in requirements, such as laws, standards, technology

changes, and market needs for future applications.

- **Domain design.** The domain design sub-process encompasses all activities for defining a *reference architecture* of the product family. The reference architecture provides a common, high-level structure for all members of the product family. In the case of embedded software systems, the reference architecture should address both hardware and software architectures, as well as their commonalities and variabilities.
- **Domain realization.** The domain realization sub-process deals with the detailed design and the implementation of reusable components. Output of this sub-process consists of loosely coupled, configurable components. Each component is planned, designed, and implemented for reuse in different contexts, i.e. members of the product family.
- **Domain testing.** After the reusable components are developed, they are tested and validated against their specifications. There is no running application to be tested in domain testing. Only single components and integrated chunks composed of common parts can be tested in domain testing. It is also possible to create and test sample products that contains some variable parts.

Application engineering sub-processes

The key goal during the application engineering is to achieve as high as possible reuse of the domain assets when defining and developing individual products. The main sub-processes of application engineering are:

- **Application requirements engineering.** During this sub-process functional requirements and quality attributes of a specific product are extracted, documented, analyzed, and linked to the domain requirements. Some of the application requirements may be left uncovered by the domain requirements. In this case, usually, the domain requirements are needed to be updated. As a result, a new increment of domain engineering is initiated.
- **Application design.** Using the application requirements the reference architecture is customized, in this sub-process, to meet the needs of the respective product. Customization is done through making configuration decisions that must comply with the rules defined in the reference architecture. Output of the application design sub-process is a

product specification describing the architecture and design of the final product. Such a specification is created using the configuration decisions. For an embedded software system, the product specification usually contains a specification of both hardware and software.

- **Application realization.** The application realization sub-process creates the considered product. During this sub-process reusable components are instantiated and assembled according to the configuration decisions.
- **Application testing.** The application testing sub-process encompass activities for testing the system created in the previous sub-process. These activities include software and hardware unit testing, as well as integration testing.

More details about domain and application engineering processes and their sub-processes can be found in [18].

3 Model-Based Configuration

The ultimate goal in this thesis is to tackle the configuration challenges that are faced, in practice, during the development of integrated control systems. In this section, we first explain the configuration challenges and their origins. Then we provide an overview of our contributions to solving the problem.

3.1 Configuration challenges

Software configuration is a major part of the application design sub-process (Figure 1). It encompasses the main activities for creating an individual member of the product family. During software configuration, reusable software components are selected and customized according to the requirements of the product. In practice, the main reusable asset of a software product family is, usually, a parameterized code-base – a large body of source code in C, C++, or Java – which can be configured through assigning values to its parameters. We refer to these parameters as *configurable parameters*, and to the values assigned to them as *configuration decisions*. Configuration engineers are responsible for making the configuration decisions.

Output of the software configuration process is a *configuration file*, which provides a (possibly partial) specification of a product. A configuration file is created from configuration decisions, and is usually very similar to a main method in C++ or Java. It contains the information for creating instances of the classes in the parameterized code-base and initializing those instances. Figure 2 shows the inputs and the output of the software configuration process.

Complete configuration files are used, during application realization, to build software products. The resulting software products are tested during application testing. Results of a study [7] that we performed at our industry partner shows that a considerable percentage (about 40%) of the errors discovered during application testing are in fact due to configuration errors. Configuration errors make application engineering a time-consuming and costly process. Configuration files are created, tested, and modified in several rounds until a valid configuration file (i.e., a configuration file that complies with the reference architecture and that satisfies the product requirements) is achieved.

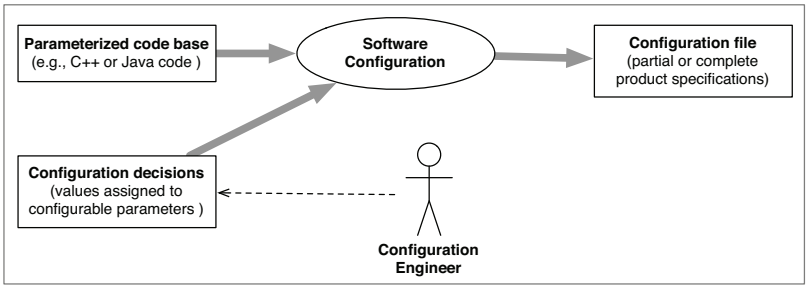


Figure 2: Software configuration in integrated control systems domain.

Devising a solution to configuration errors and the costly process of debugging configuration files requires understanding the sources of configuration errors. To obtain such an understanding, we studied error-logs and root-cause-analysis reports at our industry partner. According to the findings of our studies, which are reported in Paper1 (i.e., [7]) and are congruent with findings of previous studies reported in the literature (e.g., [10]), the main sources of configuration errors are:

- **Large number of interdependent configurable parameters.** To create a software product, several tens of thousands of parameters should be configured, manually, by the

configuration engineers. This results in extensive workload on configuration engineers and higher chances of making incorrect configuration decisions.

- **Insufficient documentation.** To configure the software, configuration engineers require information about the reference architecture, reusable components, configurable parameters and their interdependencies. Due to the scale and heterogeneity of integrated control systems and the evolution of product families, it is usually expensive for companies to maintain a concise and up-to-date documentation of their product family assets. This makes the job of configuration engineers even more difficult as they have to rely on tacit knowledge and the information scattered in various (possibly inconsistent) sources to make the configuration decisions.
- **Insufficient support for configuration validation.** Automated support for validating configuration decisions during the application design sub-process is very limited. Usually, configuration tools are incapable of validating partially specified product configurations and checking the compliance of configuration decisions with the reference architecture. Therefore, configuration validation is left, to a great extent, to the testing sub-process. This late validation of the configuration decisions makes localizing and fixing configuration errors complicated and laborious.

All these factors contribute to the configuration challenges. However, insufficient documentation contributes in two directions. First, it imposes extra work on the configuration engineers who have to seek the required information from various sources. Second, it contributes to the lack of automated support for configuration validation, as concise specifications are crucial to providing any form of automation support. Insufficient documentation is, therefore, a major challenge that should be addressed by any configuration solution.

3.2 Overview of the configuration solution

The main contributions of this thesis are the definition and the development of a model-based configuration solution that addresses the software configuration challenges described above with an emphasis on integrated control systems. We have devised and developed a model-based semi-automated configuration approach that tackles the configuration challenges

by helping configuration engineers create consistent and error-free software configurations. The idea of reuse-oriented configuration is proposed to reduce configuration effort by automatically making configuration decisions based on the internal similarities of individual products. The reuse-oriented configuration approach is developed as an extension to our semi-automated configuration approach. In parallel with these configuration approaches, we have provided a mathematical formalization for the main computations in our configuration solution. Specifically, we have formally defined the notion of configuration in our context, and specified how our solution to configuration can ensure the consistency of final products.

3.2.1 Model-Based Semi-Automated Configuration

Figure 3 shows an overview of our model-based semi-automated configuration approach. As shown in this figure our approach has two major steps. In the first step (product-line modeling step), which maps to the domain design sub-process in Figure 1, a model of a product family is created. In the second step (guided configuration step), which maps to application design sub-process in Figure 1, a configuration engine uses the product family model to provide three functions that enable consistent configuration of the software. These two steps address and enhance the product-line engineering framework's support for abstraction and automation, respectively.

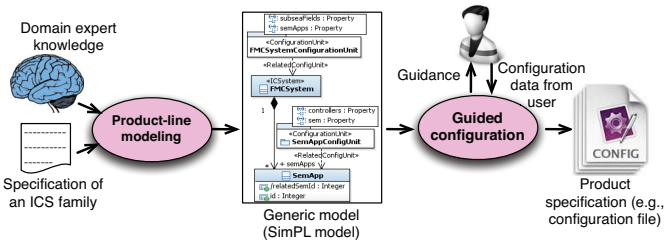


Figure 3: An overview of our model-based configuration approach.

For the product-line modeling step, we have devised and implemented the SimPL methodology. This methodology is based on modeling standards, such as UML, MARTE, and OCL, and enables engineers to create product-family models from textual specifications and domain experts knowledge. The SimPL methodology enables specifying software and

hardware components, the dependencies among them, and the variabilities in them. The design rationales and the methodology itself form the first major contribution of this thesis and are presented in Paper1.

For the guided configuration step, we have proposed a semi-automated configuration approach that iteratively and interactively collects configuration decisions from configuration engineers and creates a product specification that is consistent with the input product-family model. To ensures the consistency of the final product specification, our approach provides three functions:

- **Instant configuration validation.** Each time the user provides a configuration decision, the decision is validated against the product-family model and previously made configuration decisions. Such an instant configuration validation ensures that configuration errors are discovered as early as possible (i.e., immediately after they are made). Fixing errors at such an early stage is expected to be easier and straightforward.

- **Interactive user guidance.** To reduce chances of error, we use the information in the product-family model to guide configuration engineers throughout the configuration process and to help them make consistent configuration decisions.

- **Automated decision making.** To reduce the workload on configuration engineers, and to reduce chances of making configuration errors, we automate some of the configuration decisions. To do so, we use constraint satisfaction techniques to infer configuration decisions from the information in the product-family model and the previously made configuration decisions.

To provide these functionalities, we use constraint satisfaction techniques, especially, constraint propagation over finite domains [14]. These techniques are particularly advantageous in this context because they allow validating and exploring partially specified product configurations. The semi-automated configuration approach is the second major contribution of this thesis and is presented in details in Paper3. A formalization of the semi-automated configuration approach and a detailed specification of its functionalities in terms of constraint satisfaction operations are presented in Paper2.

3.2.2 Reuse-Oriented Configuration

Figure 4 shows an overview of our approach to reuse-oriented configuration. This approach, as shown in Figure 4, is an extension to our model-based configuration approach, where both modeling and configuration steps are extended. The idea in this extension is that by modeling the internal similarities of a product, and using the configuration engine to ensure the consistency of the final product with respect to those internal similarities, the automated decision making capability of the configuration engine can be triggered to automatically make a great portion of the configuration decisions, therefore reducing configuration effort.

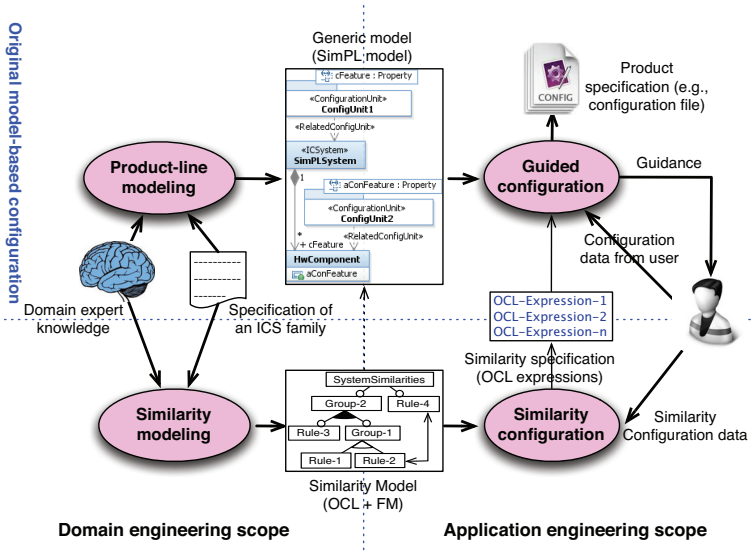


Figure 4: An overview of our reuse-oriented configuration approach.

Our reuse-oriented configuration approach has four major steps. The first step (the Product-line modeling step) is the same as that in our model-based configuration approach in Figure 3. In the second step (the Similarity modeling step), possible structural similarities that may exist in some particular products are modeled and organized in a *similarity model*. In the third step (the Similarity configuration step), the similarity model is used to generate *similarity specifications* of particular products. Finally, in the Guided configuration step, we use our semi-automated configuration approach to generate product specifications that comply both

with the generic SimPL model of the product family and with the similarity specifications of the products generated in the third step. As shown in Figure 4, the two first steps belong to the domain engineering process (more specifically, to the domain design sub-process) and are performed once per product family. The last two steps, i.e., Similarity configuration and Guided configuration steps, belong to the application engineering process (more specifically, to the application design sub-process) and are repeated for each product.

The idea of reuse-oriented configuration and our similarity modeling approach form another major contribution of this thesis. Our similarity modeling approach enables creating similarity models (the second step of Figure 4) that are used as a basis for configuration reuse. A similarity model expresses the structural similarities in two levels of abstraction. In the lower level of abstraction, OCL is used to express the similarity in terms of the model elements in the SimPL model of the product family. Each OCL constraint in this level specifies one *similarity rule*. In the higher level of abstraction, a feature model [15] is used to provide a user-level representation of the similarity rules. The reuse-oriented configuration and our similarity modeling approach are explained in details in Paper4.

3.2.3 Formal specification of the Semi-Automated Configuration Approach

A pivotal piece of the work that is done as part of this thesis is the formalization that we have provided for the notion of configuration and its consistency, in our context. As part of this formalization, we have provided definitions for consistent product family models and consistent product specifications. A mapping from product and product family specifications to a finite domains constraint program is provided. We have used this mapping to define the main functionalities of our model-based semi-automated configuration approach (Section 3.2.1) in terms of the constraint satisfaction operations over finite domains. This formalization is presented in Paper2.

4 Research Method

Several research methods have been employed in various parts of this thesis. As an industry-driven research, we started the work by understanding the industrial context to identify

and carefully define the main problems at our industry partner, FMC Technologies [1]. Characteristics of an adequate solution for the identified problems were then defined and a literature review was performed to assess the existing work. We defined and developed a modeling methodology and the automation support required for realizing the adequate solution. We conducted several experiments and empirical studies to evaluate our solution. Moreover, in parallel with developing and evaluating our solution, we provided a mathematical formalization for our approach, including specifications and proofs for major properties of our solution.

4.1 Understanding the industrial context

Software development is a time-consuming process at our industry partner and the final products are sometimes faulty. To find a solution to these problems, in this thesis, we started by a close collaboration with our industry partner to identify the sources of these problems. We had several meetings and we studied several documents, error-logs and root-cause-analysis reports. From this investigation, we found out that (1) at FMC, they have a product family for their products, and in particular, a software product family for the software they develop, and (2) about 40% of the software problems are due to configuration errors.

In the second phase of our collaboration with FMC, we preformed a domain analysis to identify the characteristics of their products, and the configuration challenges that lead to configuration errors. FMC products are large-scale integrated control systems, where software controls thousands of electrical and mechanical devices. Characteristics of FMC products and their configuration challenges are, as discussed in Paper1, generalizable to many integrated control systems.

Based on the domain analysis results, we specified a set of characteristics for an adequate configuration solution (i.e., a solution that can address the configuration challenges that are the focus of this thesis). From those characteristics, we derived a set of modeling requirements that need to be fulfilled by a model-based approach to configuration. The identified characteristics and the modeling requirements serve as a basis for the rest of the research that we did throughout this thesis. Details of our research method regarding understanding the context

and characterizing a model-based configuration solution are presented in Paper1.

4.2 Literature Review

After understanding the industrial context and configuration challenges in the context of integrated control systems, the next step was to survey the literature and assess existing solutions. To do so, we first derived a set of assessment criteria matching the modeling requirements and the identified characteristics for an adequate solution, and used them to evaluate existing approaches.

In the first phase of the literature review, we evaluated existing approaches for variability modeling. As mentioned in Paper1, none of them could fulfill all of our criteria. This is why we had to define and develop the SimPL methodology, which shares similarities with some of the existing variability modeling approaches, but specifically addresses all the modeling requirements to support the configuration of integrated control systems.

In the second phase of our literature review, we investigated and evaluated existing solutions to effective and consistent software configuration. According to our findings reported in Paper2 and Paper3, none of the approaches were capable of handling complex constraints that we have in our product-family models, and none of them supports interactive user guidance, which is a major contribution of our approach.

Finally, we could not find any work related to the reuse-oriented configuration approach.

4.3 Developing a Model-Based Configuration Approach

To fill in the variability modeling gap mentioned above, we defined and developed the SimPL methodology. This methodology is designed to fulfill the modeling requirements and a set of practicality requirements that are defined to ensure that our approach can be applicable in practice. The modeling and practicality requirements as well as the SimPL methodology are explained in Paper1. In Paper4, an extension to the SimPL methodology is proposed, which enables similarity modeling.

To fill in the gap in the automated support for configuration, we proposed and developed a model-based, semi-automate configuration approach. The three functionalities that we

provide in our semi-automated configuration approach, together with the reuse-oriented configuration approach, address the configuration challenges and fulfill the characteristics of an adequate configuration solution. Paper3 and Paper4 explain in details our solution to the configuration challenges.

4.4 Empirical Studies

As a fundamental part of this thesis we conducted several empirical studies using real-world case studies and examples from our industry partner to evaluate the capabilities of the SimPL methodology, and the semi-automated and the reuse-oriented configuration approaches. We applied the SimPL methodology to create a model of a product family that FMC delivers. Details about the product family and the model we created and our evaluations and discussions are presented in Paper1. We used two configured products from the same product family to evaluate our semi-automated configuration approach and our reuse-oriented configuration approach. These experiments and their results are presented in Paper3 and Paper4.

4.5 Mathematical Analysis

As a last piece of the work, we have provided the mathematical structures and theories underlying our configuration solution. The goal is to specify and prove the main characteristics of our approach. In particular, we prove that the product specifications that are the output of our semi-automated configuration approach are consistent with the input product family models. For this purpose, mathematical specifications for the main concepts in the SimPL methodology are provided, and the notion of configuration is formalized and redefined as a constraint program where constraint satisfaction operations can be applied to realize instant configuration validation, interactive user guidance, and automated decision making. We have also formalized the notion of consistency both at the product level and at the product family level. These definitions and the resulting theorems and proofs are presented in Paper2.

5 Summary of Results

The main research results of our work are elaborated in the four papers included in this thesis. In this sections, we summarize the key results obtained from each paper.

Paper1

SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems. *Razieh Behjati, Tao Yue, Lionel Briand, Bran Selic*. Accepted for publication in the Journal of Information and Software Technology, 2012.

This paper reports on the first part of the work in this thesis. Specifically, we provide an introduction to the industrial context and present an analysis of the configuration problems in the integrated control systems domain. We describe the characteristics of an adequate solution. To provide a model-based realization of the adequate solution, we have provided a list of modeling and practicality requirements and developed the SimPL methodology that provides the notation and guidelines for creating models that fulfill these requirements. The following are the research questions that are addressed in this paper:

- What are the main configuration challenges in the integrated control systems domain?
- What are the characteristics of an adequate solution to the configuration challenges?
- What requirements should the product-family models fulfill to enable a model-based realization of the adequate configuration solution?
- To what extent does the SimPL methodology fulfill the modeling requirements?

The paper reports that tacit knowledge and inadequate documentation, insufficient configuration guidance, lack of automated configuration validation, and insufficient support for configuration reuse are the main configuration challenges that need to be addressed by an adequate configuration solution. Such a solution should automatically validate configuration decisions, interactively guide configuration engineers throughout the configuration process, and effectively reduce configuration effort for example through automating some of the configuration decisions. In addition, the configuration solution should be complete and scalable. Completeness means that the approach should be able to collect and validate all

types of configuration decisions. Scalability means that the approach should be able to handle the large diversity in the types of reusable components as well as the large numbers of configuration decisions that are normally involved in the production of individual products.

Modeling requirements are derived based the abovementioned characteristics and express certain qualities (elaborated in Paper1) for software and hardware models, dependencies between hardware and software components, variabilities in hardware and software, and organizing such variabilities. The ability of the SimPL methodology in fulfilling these requirements is evaluated through applying it for modeling a product family from our industry partner. Results of this study show that for the subject of our study, which is representative in terms of characteristics of integrated control systems families, the SimPL methodology is powerful enough to provide a model of the product family that meets the modeling requirements.

Paper2

Architecture-Level Configuration of Large-Scale Embedded Software Systems: A Formal Specification. *Razieh Behjati, Shiva Nejati, Lionel Briand*. Submitted to ACM Transactions on Software Engineering and Methodology, 2012.

This paper is an extension of Paper3. However, we have included it as the second paper in this thesis because it provides a detailed explanation of the notion configuration, which is an instrumental concept in our work but is only briefly and intuitively discussed in Paper1 and Paper3. In this paper, we formalize the main concepts in a SimPL model that are involved in the configuration process. We provide a mapping from these concepts to the elements of a constraint program, present the configuration algorithm, define the notion of consistency, and prove that our configuration algorithm produces consistent product specifications. This paper answers three research questions:

- Is a constraint program expressive enough to model consistency-related aspects of a product specification that is derived from a SimPL model?
- Does the configuration algorithm (involving the invocation of constraint propagation algorithms) terminate?
- Are the output product specifications guaranteed to be consistent with respect to the

input product family models?

In our model-based semi-automated configuration approach, SimPL models are class-based models specifying product families. Product specifications are, on the other hand, instance based specifications where classes and associations in the SimPL model are instantiated. To answer the first question, we have provided a mapping from these class-based and instance-based specifications to the constraint programs that are the input to the constraint propagation algorithm. This paper shows that the product specification of an integrated control system can be mapped to a finite domains constraint program and the mapping takes linear time (i.e., it has a time complexity proportional to the number of the configurable parameters in the product).

Our approach to configuration is iterative. In each iteration, the configuration engineer makes a configuration decision, which is validated and, if valid, is used to provide user guidance and infer new configuration decisions. To formalize these functionalities, we have defined the notion of valid domains. For each configurable parameter, its valid domain is a finite set of values that can be assigned to that parameter without resulting in any inconsistencies. Valid domains are recomputed each time the user assigns a value to one of the configurable parameters. The main computation in each configuration iteration is the calculation of the valid domains for which we invoke the constraint propagation technique. Constraint propagation [14] is a monotonic algorithm and its termination is guaranteed. However, in our approach, certain types of configuration decisions may violate the monotonicity of the algorithm. We handle these cases differently, i.e., we recreate the constraint program and start a new constraint propagation session, to guarantee the monotonicity and the termination of the algorithm.

Finally, in this paper we have proven, using the provided formalism and the mapping to constraint programs, that our configuration algorithm guarantees the consistency of the final product specifications provided that the input family models are consistent.

Paper3

Model-Based Automated and Guided Configuration of Embedded Software Systems. *Razieh Behjati, Shiva Nejati, Tao Yue, Arnaud Gotlieb, Lionel Briand*. Eighth European Conference on Modeling Foundations and Applications (ECMFA), 2012.

This paper presents our model-based semi-automated configuration approach, details of an implementation of the configuration algorithm (presented in Paper2) using the SICStus Prolog [5, 8], and empirical results of applying the approach on industrial case studies from our industry partner. We performed several experiments with the industrial case studies to answer the following research questions:

- What percentage of the configuration decisions can be automated using our configuration approach?
- How much do the valid domains shrink at each configuration iteration?
- How long does it take to propagate a user's decision and provide guidance?

Saving a number of configuration steps through automatically making configuration decisions is expected to reduce configuration workload, and reduction of the domains decreases the complexity of decision making. Therefore, answers to the first two research questions provide insights into how much configuration effort can be saved. Our results show that our approach can automatically make up to 50% of the configuration decisions and, in average, reduces the valid domains by 40% in each iteration.

Answering the third research question provides insights into the applicability and scalability of our technique. Our results show that, in our current implementation of the configuration approach, the average time required for propagating each user decision grows quadratically with the number of configurable parameters. For real-world applications where the number of configurable parameters is in the tens of thousands, the current implementation would be inefficient and impractical. This inefficiency is in no way a drawback of our configuration approach, but a result of our current implementation, which does not take full advantage of the capabilities of constraint propagation techniques. In particular, constraint propagation allows introducing new constraints on-the-fly and recomputing only the valid domains of the involved configurable parameters. According to our observations reported in

Paper3, the degree of dependency – defined for each configurable parameter as the average number of configurable parameters related to it – is relatively low (i.e., about one thousandth of the total number of configurable parameters), which implies that very few valid domains need to be recomputed in each configuration iteration. This suggests that the on-the-fly propagation capability can be very beneficial in our context.

In our current implementation, however, we could not get benefit from the on-the-fly propagation capability because the Java interface that we use from the SICStus Prolog does not support this capability. As a result, in each configuration iteration, a new constraint propagation session is created and all the valid domains are computed from scratch by propagating all the constraints instead of propagating only the newly introduced constraints. Therefore, by improving our current implementation and using the on-the-fly constraint propagation capability, we can considerably improve the efficiency and applicability of our approach.

Paper4

A Modeling Approach to Support the Similarity-Based Reuse of Configuration Data. *Razieh Behjati, Tao Yue, Lionel Briand.* Model Driven Engineering Languages and Systems, 15th ACM/IEEE International Conference, MODELS 2012.

The promising results, obtained in Paper3, on reducing the configuration effort through automatically making configuration decisions, and the high degree of similarity that we had observed in products in the integrated control systems domain, motivated us to design an approach for automatically reusing configuration decisions based on the internal similarities that exist in integrated control systems. To do so, in this paper, we have proposed a similarity modeling approach to create, as part of the product family model, concise specifications of internal similarities that may be required in some products. Internal similarities are modeled as a set of OCL constraints each representing a similarity rule. Similarity rules can be activated or deactivated before the configuration of each individual product. Activated similarity rules specify consistency rules that require certain configurable parameters to have the same values in the product. Since our configuration approach aims at maintaining the consistency of the

product specifications, these activated similarity rules result in automated value assignment for some configurable parameters whenever a value is assigned to a similar configurable parameter. We designed and performed an experiment to answer the question: "What percentage of the configuration decisions can be automated based on internal similarities for full product specifications?" Results of our experiment shows that more than 60% of the configuration decisions can be reused using this approach.

Internal similarities are variable features of the product family. Different members of a family may require different similarity rules to be activated and applied. We have implemented this in our approach using feature models and their configuration. As expected, results of our experiments showed that different degrees of internal similarity can significantly affect the percentage of configuration reuse.

6 Directions for Future Work

The research presented in this thesis suggests three directions for future work. First, our approach to providing interactive user guidance can be extended to provide other forms of guidance. The results of our experiments presented in Paper3 shows that the order of decision making can affect (1) the percentage of configuration decisions that can be automated, and (2) the reduction of the valid domains. By finding the optimal ordering and providing that to configuration engineers as a form of user guidance, we can increase the percentage of automated configuration decisions, therefore, further reducing the configuration effort. Moreover, making the configuration decisions according to such optimal ordering can increase the reduction of the valid domains, therefore reducing the complexities of decision making. The first direction for future work would be to devise approaches and heuristics for efficiently deriving the optimal ordering.

A second direction for future work concentrates on improving the efficiency of our semi-automated configuration approach. As mentioned earlier, improvements are required in our current implementation to reduce the time required for propagating configuration decisions and recomputing the valid domains in each iteration. Moreover, in the future, we will devise

heuristics for localizing the propagation of configuration decisions to further improve the performance of our approach, which is very crucial for it to be applicable in practice.

Finally, we will perform several experiments with human subjects, mainly to assess ease of use and applicability of our SimPL methodology and our similarity modeling approach. Both the semi-automated configuration approach and the reuse-oriented configuration approach gain their power from the underlying models. Therefore, solutions, including plug-ins and tool supports, that can facilitate creating these models should be devised and implemented. Experiments with human subjects can serve as a first step to that end.

7 Conclusion

For families of integrated control systems (ICSs) the configuration process is a time-consuming and error-prone task that is complicated by several factors. Large numbers of interdependent configurable parameters together with insufficient automation support and the lack of systematic, complete, and up-to-date documentation results in a higher chance for human errors. The objective of our research is to provide an applicable configuration solution to address the configuration challenges present in the ICSs domain. Such a solution is expected to guide configuration engineers throughout the configuration process, automatically verify configuration decisions, partially automate decision making, and support configuration reuse, with the goal of improving the overall quality and productivity of the software configuration process.

As the first step to achieve the above objective, we proposed in Paper1 a modeling methodology, named SimPL, for creating product-family models that can be used as a foundation for semi-automated configuration of ICSs. Models created based on SimPL mainly target at capturing reusable components, their configurable parameters, and the dependencies between them. The need for a new modeling methodology was justified to meet a set of modeling and practicality requirements derived from a careful analysis of ICS families, their configuration challenges, and the characteristics of an adequate configuration solution in this context. An analysis of the existing work in the literature shows that none of the existing

approaches fulfill all of these requirements. SimPL is a methodology that is specifically designed to meet them and shows to do so based on the results of an industrial case study we performed.

As the second step, we presented, in Paper3, an automated model-based configuration approach for large-scale embedded software systems. Our approach builds on the SimPL methodology, and uses constraint solvers to interactively guide engineers in building and validating software configurations. We evaluated our semi-automated configuration approach by applying it to a real ICS family from our industry partner where we rebuilt three verified configurations to evaluate three important practical factors: (1) reduction in configuration effort, (2) reduction in the probability of human errors, and (3) scalability. Our evaluation showed that, for the subjects of the experiment, our approach (1) can automatically make up to 50% of the configuration decisions, (2) can reduce the complexity of decision making by 40%, and (3) can, in average, evaluate each configuration decision in less than 9 seconds.

While our preliminary evaluations demonstrate the effectiveness of our approach, the value of our tool is likely to depend on its scalability to very large and complex configurable systems. In particular, being an interactive tool, its usability and adoption will very much depend on how fast it can provide relevant guidance information at each iteration. Our current analysis shows that the propagation time grows polynomially with the size of the product. But this seems to be a result of our current implementation, which does not take full advantage of the capabilities of constraint propagation techniques. In the future, we will improve our implementation of the configuration approach, and by doing so we expect to considerably improve the efficiency and scalability of our approach.

Individual ICS products, like many other embedded software systems, usually bear a high degree of similarity within their hardware structures, which results in internal similarities within their software configurations. The third step, presented in Paper4, focuses on the automated similarity-based reuse of configuration data based on such internal similarities. We have proposed a similarity modeling approach to capture such internal similarities. Internal similarities are specified in terms of the elements in the SimPL model of the product family as a set of similarity rules using OCL. Each similarity rule can be seen as an optional feature of the product family, therefore introducing a point of variability. We use feature models to

provide a user-level representation of similarity rules and the variabilities they introduce.

We evaluated the effectiveness of our approach using two product configurations from our industry partner. Our results show that an automated similarity-based approach to configuration reuse can save more than 60% of configuration decisions, and consequently, can significantly reduce configuration effort. In the future, we will conduct experiments with human subjects, to further evaluate the applicability of our approach.

Bibliography

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

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

[3] MARTE: Modeling and Analysis of Real-Time and Embedded Systems. http://www.omgmarte.org/, 2012.

[4] OCL: Object Constraint Language. http://www.omg.org/spec/OCL/2.2/, 2012.

[5] SICStus Prolog. `www.sics.se/sicstus/`, February 2012.

[6] SysML: OMG Systems Modeling Language. http://www.omgsysml.org/, 2012.

[7] R. Behjati, T. Yue, L. Briand, and B. Selic. SimPL: a product-line modeling methodology for families of integrated control systems. *Accepted for publication in Inf. Softw. Technol.*, 2012.

[8] M. Carlsson and P. Mildner. SICStus Prolog – the first 25 years. *CoRR*, abs/1011.5640, 2010.

[9] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP*, 1997.

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

[11] S. Friedenthal, A. Moore, and R. Steiner. *A Practical Guide to SysML: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., 2008.

[12] D. Gasevic, D. Djuric, and V. Devedzic. *Model Driven Engineering and Ontology Development*. Springer Publishing Company, Incorporated, 2nd edition, 2009.

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

[14] P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). In *Selected Papers from Constraint Programming: Basics and Trends*, 1995.

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

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

[17] G. Perrouin, J. Klein, N. Guelfi, and J. M. Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC*, 2008.

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

[19] R. Rabiser, P. Grünbacher, and D. Dhungana. Supporting product derivation by adapting and augmenting variability models. In *SPLC*, 2007.

[20] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Inf. Softw. Technol.*, 52(3), March 2010.

Paper 1:
SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems

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

Razieh Behjati[1,2], **Tao Yue**[1], **Lionel Briand**[1,2,3], **Bran Selic**[1,4]

[1] Certus Verification and Validation Center, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

[3] SnT Centre, University of Luxembourg
Luxembourg

[4] Malina Software Corp.
Ottawa, Canada

Abstract:

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.

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 [1–4] 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* [5]. 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 [6], 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 paper, 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[2], 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.

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

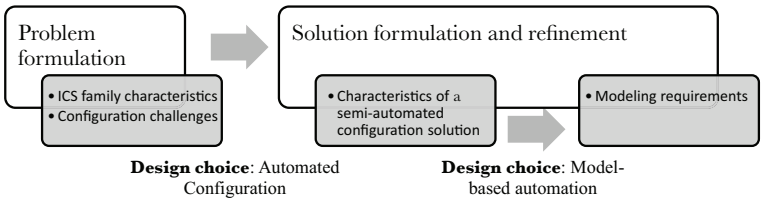


Figure 1: The engineering design process that we followed in our research.

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 [7]. 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.

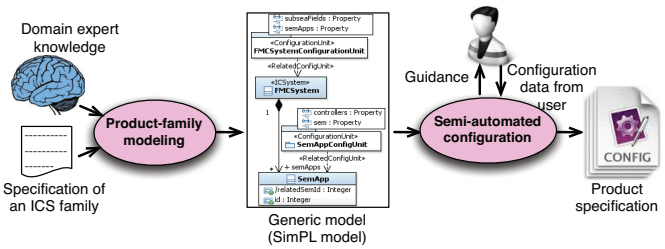


Figure 2: An overview of our model-based and semi-automated configuration solution.

The second step in Figure 2 (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 [6].

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:

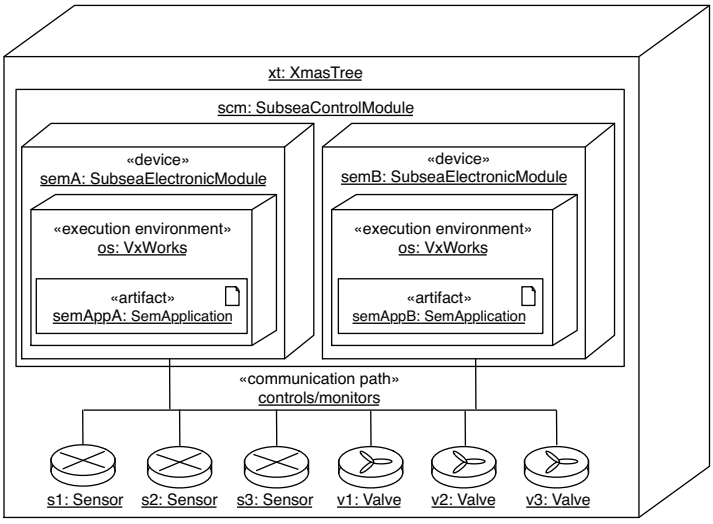


Figure 3: A simplified model of a subsea control module.

IC1. **Heterogeneous, large-scale, and hierarchical systems.** ICSs are heterogeneous sys-

tems 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 [8]. 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 [9]. 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 [9–11], 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 paper, 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 paper 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 paper 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.

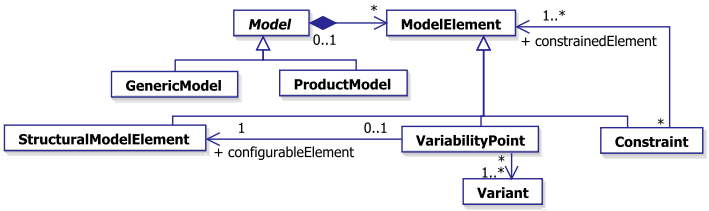


Figure 4: The conceptual model of a model-based configuration solution.

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 [12][3]. Variability points, in our context, can be instantiated[4]. 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

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

[4] 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.

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 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 [13], (2) four stereotypes from the standard MARTE profile [14], 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., [15], also see [16]).

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.

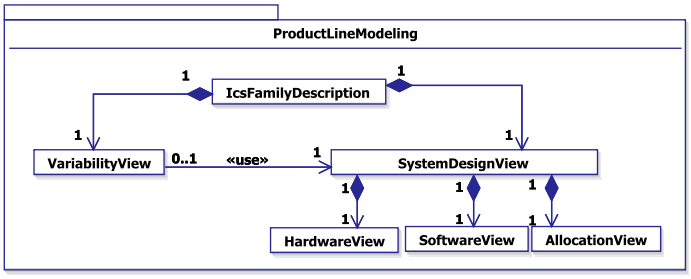


Figure 5: Multiple views introduced by the SimPL methodology.

Figure 5 is an excerpt of the SimPL metamodel[5] 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 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

[5]The complete metamodel of the SimPL methodology can be found in [17].

(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 paper, *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 [17].

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 [17]. 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[6].

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*.

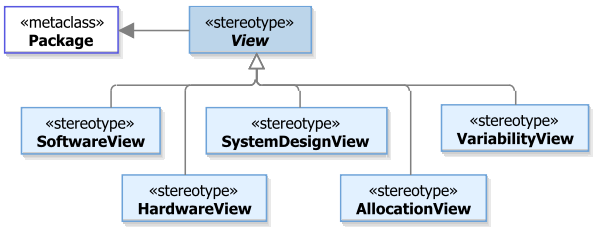


Figure 6: Viewpoint stereotypes (constrained stereotypes used to implement the SimPL methodology as a multi-view methodology).

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

[6]For this purpose, the modeling tool must allow validating profiled models and providing feedback on their validation. Some tools, e.g., IBM RSA 8 [18], provide this functionally, and are used in practice to enforce consistent use of a profile [19].

viewpoint stereotypes are presented in our technical report [17] along with the specifications of each viewpoint.

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.

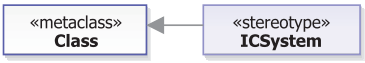


Figure 7: The «ICSystem» stereotype for denoting the topmost element.

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.

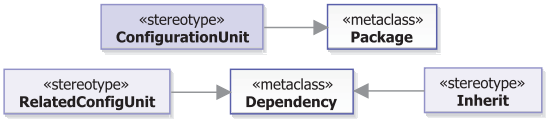


Figure 8: Stereotypes supporting variability modeling.

«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 building 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[7]. 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

[7]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.

each of these packages to provide a finer-grained organization of the model.

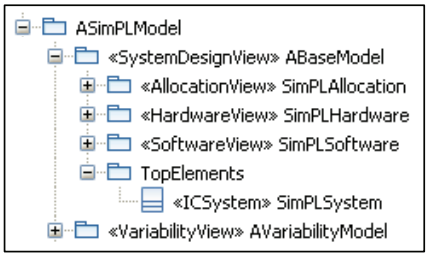


Figure 9: Sample package structure of a SimPL model.

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.

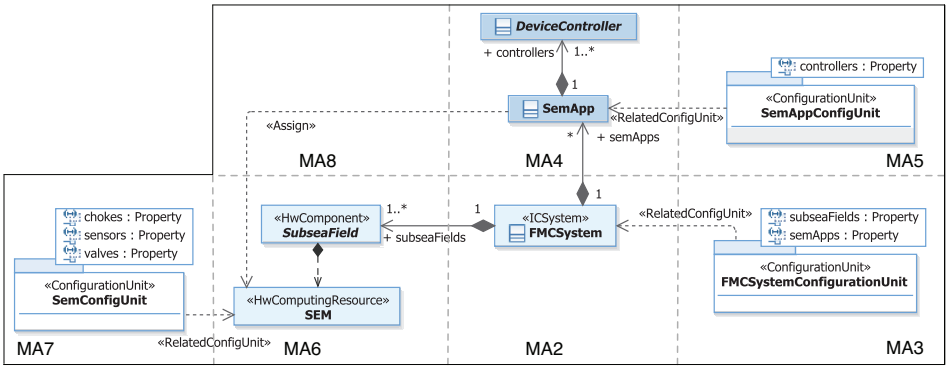


Figure 10: A summary of the main modeling activities.

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.

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[8]. 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.

[8]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).

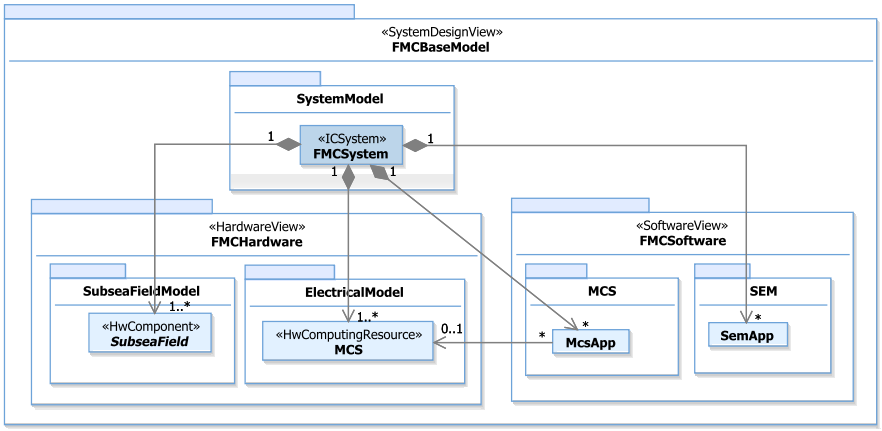


Figure 11: An excerpt of the product-line model for FMC subsea systems representing the topmost components and their relationships.

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).

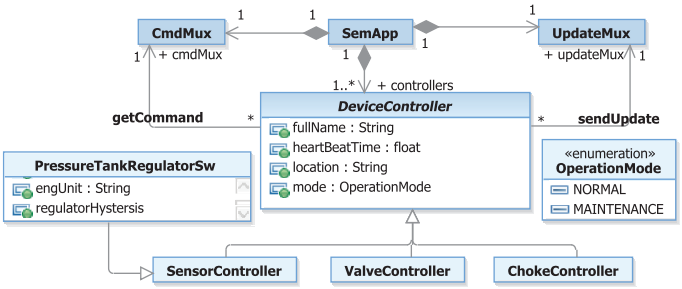


Figure 12: Decomposition of the SEM application into its subcomponents.

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.

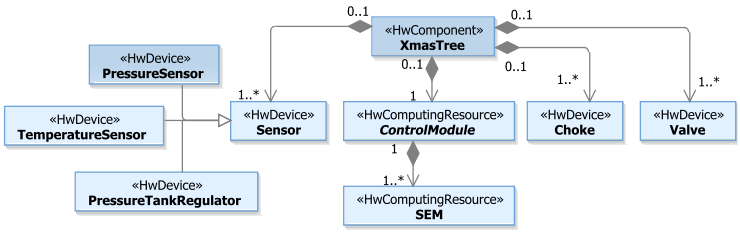


Figure 13: A model in the hardware sub-view showing the decomposition of XmasTree into its subcomponents.

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.

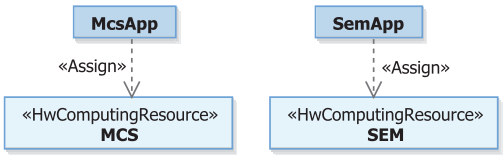


Figure 14: An example of allocation sub-view.

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.

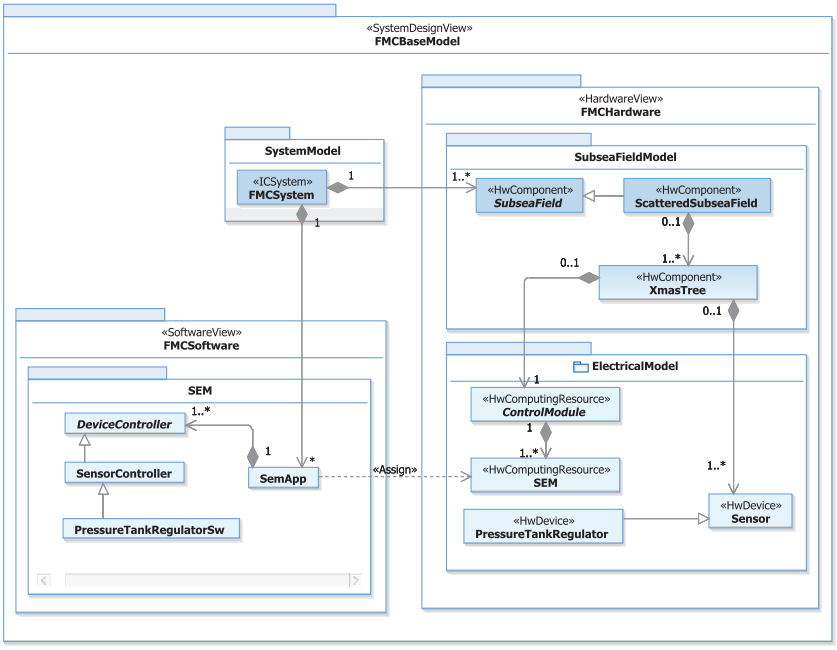


Figure 15: An excerpt of the FMC family model created by following SimPL.

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., FMCSoftware).

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) [20], 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) [21], including both structural and behavioral variabilities. However, variabilities in the behavior of software are usually handled through parameterizing

the generic code base[9]. 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

[9]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.

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.

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.

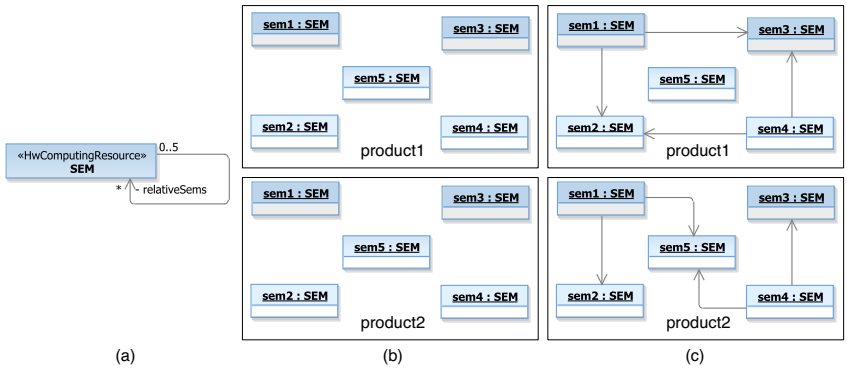


Figure 16: An example of topology variability.

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 param-

eterization, 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.

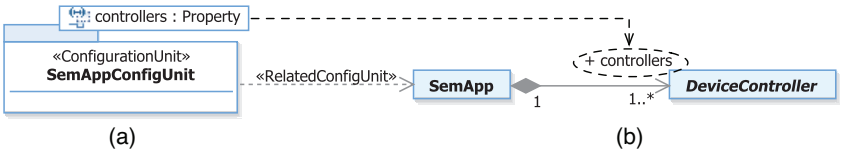


Figure 17: An example of UML template concepts and the use of «ConfigurationUnit» and «RelatedConfigUnit».

In SimPL, we use a simple form of UML *package templates* (i.e., the template specification in a template package [13]) 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 [13], 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

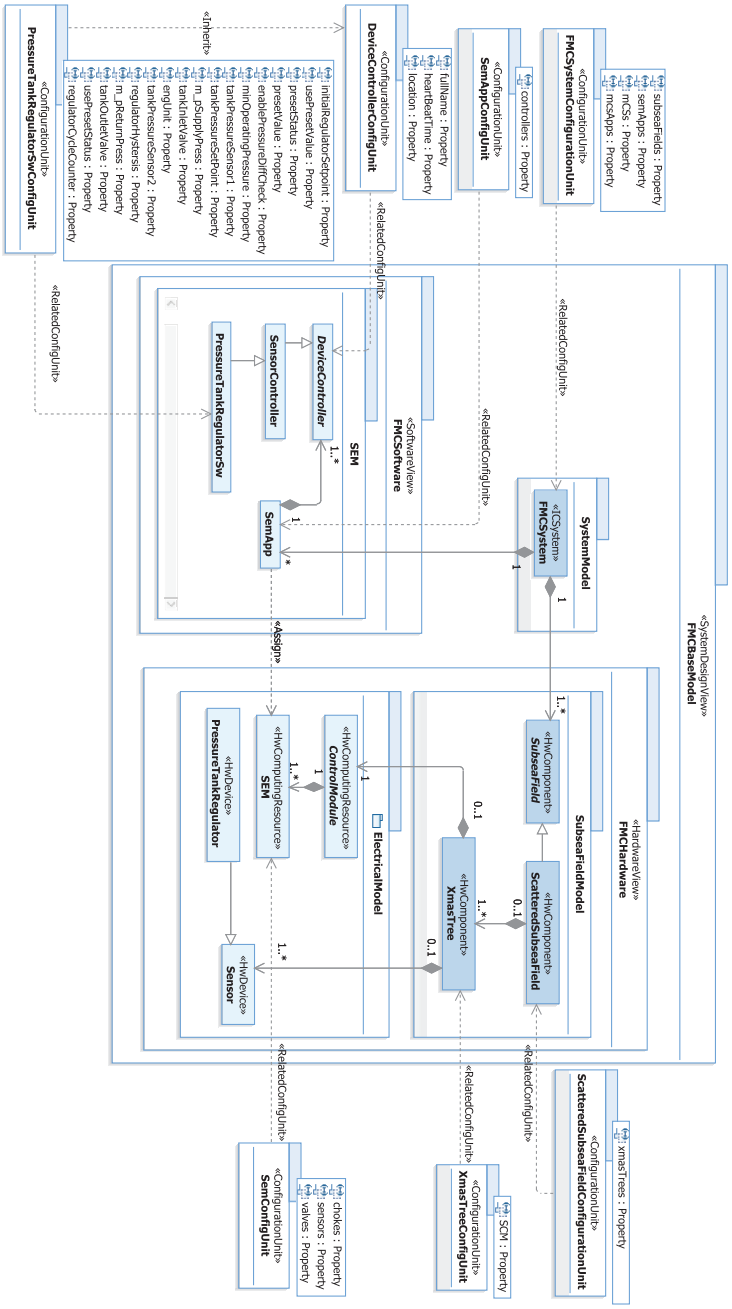


Figure 18: Extension to the class diagram in Figure 15, showing the variability points and configuration units.

the variability view. This class diagram shows seven configuration units (i.e., FMCSystem-ConfigurationUnit, SemAppConfigUnit, DeviceControllerConfigUnit, PressureTankRegulator-SwConfigUnit, ScatteredSubseaFieldConfigUnit, 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 [6].

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.

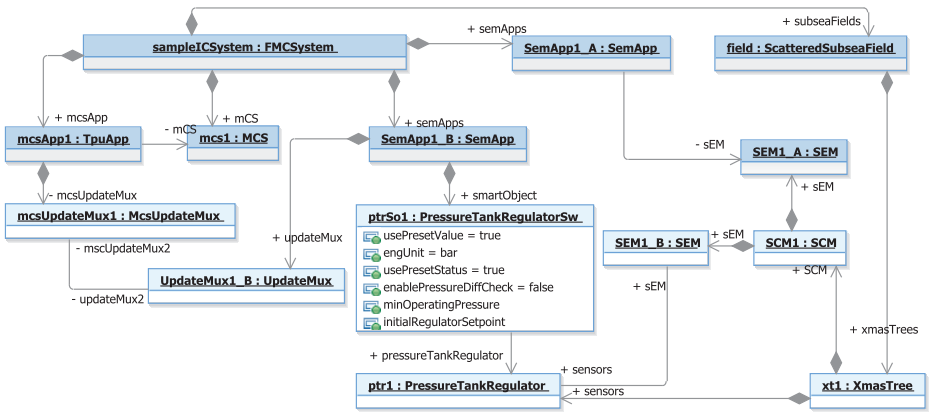


Figure 19: An example product specification.

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.

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 [6] and [22]. In particular, results reported in [6] 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 [22]. 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 [6] and [22] 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[10]. 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

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

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 [18] 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. 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 [17].

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[11]. 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

[11]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.

their origin classes.

7.2 Discussion

The experiments described in [6], 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 [6]. 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 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 [18]) 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 [23]. 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., [23, 24]) 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 [25] models product families only at the highest level of abstraction. In this paper, 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 [26–28] 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 [28, 29], as expressive as UML class diagrams and can be used to model variabilities at the architecture level.

Alternatively, several approaches (e.g., [30–32]) 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 [23] 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 [24, 33], or using domain specific languages (DSLs) as in [34, 35]. In approaches that use UML to describe base models, usually UML inherent features such as templates (e.g., in [24]) or stereotypes and profiles (e.g., in [33]) 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) [34], and approaches based on aspect-oriented modeling (AOM) [36].

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., [35]) 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 [34]. An amalgamated language is formed by the base and variability modeling languages being combined into one language (e.g., UML-based approaches such as [24, 33]), 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

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. [32]	UML / annotation	Partially supported	Supported	Not addressed	Partially supported	Not supported
Santos et al. [24]	UML / UML templates	Partially supported	Supported	Not addressed	Partially supported	Supported
Gomaa et al. [30]	UML / UML profiles	Partially supported	Supported	Not addressed	Partially supported	Not supported
Ziadi et al. [33]	UML / UML profiles	Partially supported	Supported	Not addressed	Partially supported	Not addressed
Haugen et al. [34]	DSL / CVL	Partially supported	Depends on the base modeling language		Partially supported	Partially supported
Morin et al. [35]	DSL / Aspect models	Partially supported	Depends on the base modeling language		Partially supported	Not supported
Czarnecki et al. [28]	-/ EFM	Partially supported	No separated base model			Partially supported
Sinnema et al. [23]	-/ CVVL	Partially supported	No separated base model			Supported

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., [33, 34]) 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., [24, 30–33]); 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., [24, 30, 32, 33]) can be extended using, for example MARTE [14] to support hardware modeling. Also, variability modeling approaches such as CVL can be combined with DSLs (e.g., [37]) 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., [23]) and the extended feature models (i.e., [28]) 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 [30, 33, 35], or provide other mechanisms for tracing variability points back to the base models [24, 32, 34]. 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 explic-

itly 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 [34], [24], [28], and [23].

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 paper 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 [6] to configure two ICS products. Results show that, using a SimPL model, the configuration tool can provide the automation described in this paper.

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.

Bibliography

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

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

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

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

[5] C. W. Krueger, Software reuse, ACM Comput. Surv. 24 (1992).

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

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

[8] T. Yue, L. C. Briand, B. Selic, 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.

[9] S. Deelstra, M. Sinnema, J. Bosch, Product derivation in software product families: a case study, J. Syst. Softw. 74 (2005).

[10] D. Dhungana, T. Neumayer, P. Grünbacher, R. Rabiser, Supporting evolution in model-based product line engineering, in: SPLC, IEEE Computer Society, 2008, pp. 319–328.

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

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

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

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

[15] A. Egyed, Instant consistency checking for the UML, in: ICSE, ACM, 2006, pp. 381–390.

[16] M. Usman, A. Nadeem, T. Kim, E. Cho, A survey of consistency checking techniques for UML models, in: ASEA, IEEE Computer Society, 2008, pp. 57–62.

[17] R. Behjati, T. Yue, L. C. Briand, B. Selic, SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems, Technical Report Simula/TR-2011-14, 2011.

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

[19] R. K. Panesar-Walawege, M. Sabetzadeh, L. C. Briand, Using UML profiles for sector-specific tailoring of safety evidence information, in: ER, Springer-Verlag, 2011, pp. 362–378.

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

[21] M. Becker, Towards a general model of variability in product families, in: Workshop on Software Variability Management, 2003.

[22] R. Behjati, T. Yue, L. C. Briand, A modeling approach to support the similarity-based reuse of configuration data, in: MoDELS, Springer, 2012.

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

[24] A. L. Santos, K. Koskimies, A. Lopes, A model-driven approach to variability management in product-line engineering, Nordic J. of Computing 13 (2006).

[25] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Technical Report CMU/SEI-90-TR-21, 1990.

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

[27] K. Czarnecki, S. Helsen, U. W. Eisenecker, Formalizing cardinality-based feature models and their specialization, Software Process: Improvement and Practice 10 (2005) 7–29.

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

[29] M. Stephan, M. 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, 2008.

[30] H. Gomaa, M. E. Shin, Automated software product line engineering and product derivation, in: HICSS, IEEE Computer Society, 2007, pp. 285–294.

[31] K. Czarnecki, K. Pietroszek, Verifying feature-based model templates against well-formedness ocl constraints, in: GPCE, ACM, 2006, pp. 211–220.

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

[33] T. Ziadi, J. M. Jézéquel, Software product line engineering with the UML: Deriving products, Software Product Lines (2006).

[34] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, A. Svendsen, Adding standardized variability to domain specific languages, in: SPLC, IEEE Computer Society, 2008, pp. 139–148.

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

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

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

Paper 2:
Architecture-Level Configuration of Large-Scale Embedded Software Systems: A Formal Specification

Paper 3:
Model-Based Automated and Guided Configuration of Embedded Software Systems

Model-Based Automated and Guided Configuration of Embedded Software Systems

Razieh Behjati[1], Shiva Nejati[1], Tao Yue[1], Arnaud Gotlieb[1], Lionel Briand[1,2,3]

[1] Certus Verification and Validation Center, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

[3] SnT Centre, University of Luxembourg
Luxembourg

Abstract:

Configuring Integrated Control Systems (ICSs) is largely manual, time-consuming and error-prone. In this paper, we propose a model-based configuration approach that interactively guides engineers to configure software embedded in ICSs. Our approach verifies engineers' decisions at each configuration iteration, and further, automates some of the decisions. We use a constraint solver, SICStus Prolog, to automatically infer configuration decisions and to ensure the consistency of configuration data. We evaluated our approach by applying it to a real subsea oil production system. Specifically, we rebuilt a number of existing verified product configurations of our industry partner. Our experience shows that our approach successfully enforces consistency of configurations, can automatically infer up to 50% of the configuration decisions, and reduces the complexity of making configuration decisions.

1 Introduction

Modern society is increasingly dependent on embedded software systems such as Integrated Control Systems (ICSs). Examples of ICSs include industrial robots, process plants, and oil and gas production platforms. Many ICS producers follow a product-line engineering approach to develop the software embedded in their systems. They typically build a generic software that needs to be configured for each product according to the product's hardware architecture [5]. For example, in the oil and gas domain, embedded software needs to be configured for various field layouts (e.g., from single satellite wells to large multiple sites), for individual devices' properties (e.g., specific sensor resolution and scale levels), and for communication protocols with hardware devices.

Software configuration in ICSs is complicated by a number of factors. Embedded software systems in ICSs have typically very large configuration spaces, and their configuration requires precise knowledge about hardware design and specification. The engineers have to manually assign values to tens of thousands of configurable parameters, while accounting for constraints and dependencies between the parameters. This results in many configuration errors. Finally, the hardware and software configuration processes are often isolated from one another. Hence, many configuration errors are detected very late and only after the integration of software and hardware.

Software configuration has been previously studied in the area of software product lines [20], where support for configuration largely concentrates on resolving high-level variabilities in feature models and their extensions [12, 13, 18], e.g., the variabilities specified for end-users at the requirements-level. Feature models, however, are not easily amenable to capturing all kinds of variabilities and hardware-software dependencies in embedded systems. Furthermore, existing configuration approaches either do not particularly focus on interactively guiding engineers or verifying partial configurations [6, 19], or their notion of configuration and their underlying mechanism are different from ours, and hence, not directly applicable to our problem domain [14, 16].

Contributions. We propose a model-based approach that helps engineers create consistent and error-free software configurations for ICSs. In our work, a large amount of the

data characterizing a software configuration for a particular product is already implied by the hardware architecture of that product. Our goal is, then, to help engineers assign this data to appropriate configurable parameters while maintaining the consistency of the configuration, and reducing the potential for human errors. Specifically, our approach (1) interactively guides engineers to make configuration decisions and automates some of the decisions, and (2) iteratively verifies software and hardware configuration consistency. We evaluated our approach by applying it to a subsea oil production system. Our experiments show that our approach can provide certain types of user guidance in an efficient manner, and can automate up to 50% of configuration decisions for the subjects in our experiment, therefore helping save significant configuration effort and avoid configuration errors.

In Section 2 we motivate the work and formulate the problem by explaining the current practice in configuring ICSs. We give an overview of our model-based solution in Section 3. SimPL methodology [5] for modeling families of ICSs is briefly presented in Section 4. We present our model-based approach to the abovementioned configuration problems in Section 5. An implementation of our approach as a prototype tool is presented in Section 6. An evaluation of the approach using our prototype tool is given in Section 7. In Section 8, we analyze the related work. Finally we conclude the paper in Section 9.

2 Configuration of ICSs: Practice and Problem Definition

Figure 1 shows a simplified model of a fragment of a subsea production system produced by our industry partner. As shown in the figure, products are composed of mechanical, electrical, and software components. Our industry partner, similar to most companies producing ICSs, has a generic product that is configured to meet the needs of different customers. For example, different customers may require products with different numbers of subsea Xmas trees. A subsea Xmas tree in a subsea oil production system provides mechanical, electrical, and software components for controlling and monitoring a subsea well.

Product configuration is an essential activity in ICS development. It involves configuration of both software and hardware components. Currently, software and hardware configuration is performed separately in two different departments within our industry partner.

In the rest of this paper, whenever clear from the context, we use *configuration* to refer either to the configuration process or to the description of a configured artifact.

The software configuration is done in a top-down manner where the configuration engineer starts from the higher-level components and determines the type and the number of their constituent (sub)components. Some components are invariant across different products, and some have parameters whose values differ from one product to another. The latter group, known as *configurable components*, may need to be, further, decomposed and configured. The configuration stops once the type and the number of all the components and the values of their configurable parameters are given.

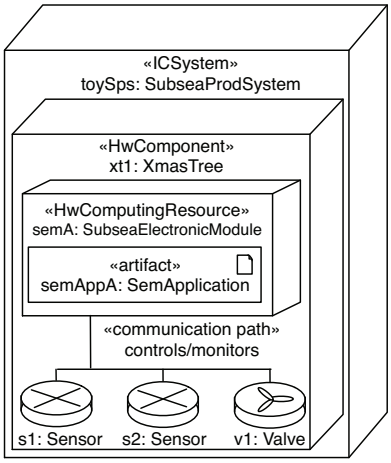


Figure 1: A fragment of a simplified subsea production system.

For example, software configuration for a family of subsea production systems starts by identifying the number and locations of SemApplication instances. Each instance is then configured according to the number, type, and other details of devices that it controls and monitors. To do this, the configuration engineer (the person who does the configuration) is typically provided with a hardware configuration plan. However, she has to manually check if the resulting software configuration conforms to the given hardware plan, and that it respects all the software consistency rules as well. In the presence of large numbers of interdependent configurable parameters this can become tedious and error-prone. In particular, due to lack of instant configuration checking, human errors such as incorrectly entered configuration data are usually discovered very late in the development life-cycle, making localizing and fixing such errors unnecessarily costly.

In short, the existing configuration support at our industry partner faces the following challenges (which seem to be generalizable to many other ICSs [5]): (1) Checking the consistency between hardware and software configurations is not automated. (2) Verification of partially-specified configurations to enable instant configuration checking is not supported. (3) Engineers are not provided with sufficient interactive guidance throughout the configuration

process. In our previous work [5], we proposed a modeling methodology to properly capture and document, among other things, the software-hardware dependencies and consistency rules. In this paper, we build on our previous work to develop an automated guided configuration tool that addresses all the above-mentioned challenges.

3 Overview of our approach

Figure 2 shows an overview of our automated model-based configuration approach. In the first step, we build a configurable and generic model for an ICS family (the Product-line modeling step). In the second step, the Guided configuration step, we interactively guide users to generate the specification of particular products complying with the generic model built in the first step.

During the product-line modeling step, we provide domain experts with a UML/MARTE-based methodology, called SimPL [5], to manually create a product-line model describing an ICS family. The SimPL methodology enables engineers to create product line models from textual specifications and the scattered domain experts knowledge. These models can then be utilized to automate the configuration process. They include both software and hardware aspects as well as the dependencies among them. The dependencies are critical to effective configuration. Currently, most of these dependencies exist as tacit knowledge shared by a small number of domain experts, and only a fraction of them, mostly those related to software, have been implemented in the existing tool used by our industrial partner. Our domain analysis [5], however, showed that failure to capturing all the dependencies have led to critical configuration errors. We briefly describe and illustrate the SimPL methodology in Section 4.

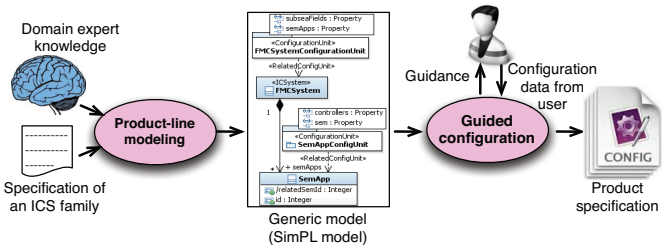


Figure 2: An overview of our configuration approach.

During the configuration step, engineers create full or partial product specifications by resolving variabilities in a product-line model. In our work, configuration is carried out iteratively, allowing engineers to create and validate partial product specifications, and interactively, guiding engineers to make decisions at each iteration. Therefore, our approach alleviates two shortcomings of the existing tool discussed in Section 2. Our configuration mechanism enables engineers to resolve variabilities in such a way that all the constraints and dependencies are preserved. At each iteration, the engineer resolves some of the variabilities by assigning values to selected configurable parameters. Our configuration engine, which is implemented using a constraint solver, automatically evaluates the engineer's decisions and informs her about the impacts of her decision on the yet-to-be-resolved variabilities, hence, guiding her to proceed with another round of configuration. In Sections 5 and 6, we describe in details how the configuration step is designed and implemented, respectively.

4 Product-line modeling

In the first step of our approach in Figure 2, we use the SimPL modeling methodology [5] to create a generic model of an ICS family. The SimPL methodology enables engineers to create architecture models of ICS families that encompass, among other things, information about variability points in ICS families.

The SimPL methodology organizes a product-line model into two main views: the *system design view*, and the *variability view*. The system design view presents both hardware and software entities of the system and their relationships using the UML class diagram notation [1]. Classes, in this view, represent software or hardware entities distinguished by MARTE stereotypes [2]. The dependencies and constraints not expressible in class diagrams are captured by OCL constraints [3]. The variability view, on the other hand, captures the set of system variabilities using a collection of *template* packages. Each template package represents a *configuration unit* and is related to exactly one class in the system design view. Template parameters of each template package in the variability view are related to the configurable properties of the class related to that package. Template packages and template parameters are inherent features in UML and are intended to be used for the specification of

generic structures. In the remainder of this section, we first describe a small fragment of a subsea product-line model, which is used as our running example. Then, using our running example, we provide a model-based view on the essential configuration activities mentioned in Section 2.

4.1 A subsea product-line model

Figure 3 shows a fragment of the SimPL model for a subsea production system[15], SubseaProdSystem. In a subsea production system, the main computation resources are the Subsea Electronic Modules (SEMs), which provide electronics, execution platforms, and the software required for controlling subsea devices. SEMs and Devices are contained by XmasTrees. Devices controlled by each SEM are connected to the electronic boards of that SEM. The electronic boards are categorized into four different types based on their number of pins. Software deployed on a SEM, referred to as SemAPP, is responsible for controlling and monitoring the devices connected to that SEM. SemAPP is composed of a number of DeviceControllers, which is a software class responsible for communicating with, and controlling or monitoring a particular device. The system design view in Figure 3 represents the elements and the relationships we discussed above.

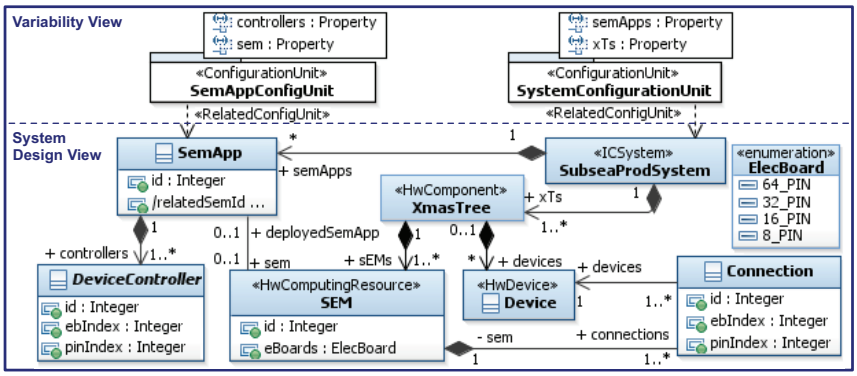


Figure 3: A fragment of the SimPL model for the subsea production system.

The variability view in the SimPL methodology is a collection of template packages.

[15] This is a sanitized fragment of a subsea production case study. For a complete model, see [5].

The upper part in Figure 3 shows a fragment of the variability view for the subsea production system. Due to the lack of space we have shown only two template packages in the figure. As shown in the figure, the package SystemConfigurationUnit represents the configuration unit related to the class SubseaProdSystem in the system design view. Template parameters of this package specify the configuration parameters of the subsea production system, which are: the number of XmasTrees, and SEM applications (semApps). Some of the other configurable parameters in Figure 3 are: the number and type of device controllers in a SemAPP as shown in SemAppConfigUnit using the template parameter controllers, the number of SEMs and devices in a XmasTree, etc.

As mentioned earlier, the SimPL model may include OCL constraints as well. Two example OCL constraints related to the model in Figure 3 are given below.

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


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

The first constraint states that the value of the pinIndex of each device-to-SEM connection must be valid, i.e., the pinIndex of a connection between a device and a SEM cannot exceed the number of pins of the electronic board through which the device is connected to its SEM. The second constraint specifies the valid range for the ebIndex of each device-to-SEM connection, i.e., the ebIndex of a connection between a device and a SEM cannot exceed the number of the electronic boards on its SEM.

4.2 Configuration activities in a model-based context

As mentioned in Section 2, configuration involves a sequence of two basic activities: (1) specifying the type and the number of (sub)components, and (2) determining the values for the configurable parameters of each component, while satisfying the constraints and dependencies between the parameters. We ground our configuration approach on the SimPL methodology

and redefine the notion of configuration in modeling terms as follows: Given a SimPL model, configuration is creating an instance model (i.e., product specification in Figure 2) conforming to the classes, the dependencies between classes, and the OCL constraints specified in that SimPL model. Such instance model is built via two activities (1) creating instances for classes that correspond to configurable components, and (2) assigning values to the configurable parameters of those instances. For example, to configure the subsea system in Figure 3, we need to first create instances of XmasTree, SEM, Device, and SemApp, and then assign appropriate values to the configurable variables of these instances. Note that value assignment may imply instance creation as well. Specifically, a configurable parameter can represent the cardinality of an association. Assigning a value to such a parameter automatically implies creation of a number of instances to reach the specified cardinality.

5 Interactive model-based guided configuration

The outcome of the configuration step in Figure 2 is a (possibly partial) model of a product that is *consistent* with the SimPL model describing the product family to which that product belongs. In our approach, SimPL models are described using class-based models, while the product models are object-based. A product model is consistent with its related SimPL model when:

- Each object in the product model is an instance of a class in the SimPL model.
- Two objects of types C_1 and C_2 are connected only if there is an association between classes C_1 and C_2 in the SimPL model.
- The object model satisfies the OCL constraints of the SimPL model.

The above *consistency rules* are invariant throughout our configuration process, i.e., they hold at each configuration iteration even when the product model is defined partially. In this section, we first describe how our approach guides the user at each configuration iteration while ensuring that the above rules are not violated. We then demonstrate how a constraint solver can be used to maintain the consistency rules throughout the entire configuration process, and to automatically perform some of the configuration iterations.

5.1 Guided and automated configuration

The product configuration process is a sequence of *value-assignment* steps. At each step, a value is assigned to one *configurable parameter*. A configurable parameter can represent (1) a property in an instance of a class, (2) the size of a collection of objects in an instance of a class, or (3) the concrete type of an instance.

A *configuration* is a collection of value-assignments, from which a full or partial product model can be generated. A configuration is complete when all the configurable parameters are assigned a specific value, and is partial otherwise. Each configurable parameter has a *valid domain* that identifies the set of all values that can be assigned to that configurable parameter without violating any consistency rule. Below, we describe the guidance information that our tool provides to the user at each iteration of the configuration process.

Valid domains. At each iteration, the tool provides the user with the valid domains for all the configurable parameters. Such domains are dynamically recomputed given previous iterations. The values that the user provides should be within these valid domains, or otherwise, the user's decision is rejected and he receives an error message. For example, the valid domain for the configurable parameter pinIndex is initially 0..63. Therefore, if a user assigns to this parameter a value outside 0..63 his decision will be rejected.

Decision impacts. If the user's decision is correct, the decision is propagated through the configuration to identify its impacts on the valid domains of other configurable parameters. This may result in pruning some values from the valid domains of some configurable parameters. For example, the valid domain for the type of an eBoard in a SEM is initially {8_PIN, 16_PIN, 32_PIN, 64_PIN} (the set of all literals in the enumeration ElecBoard). If a user configures a Connection in a SEM by assigning 2 to ebIndex, and 13 to pinIndex, then according to the OCL invariant PinRange (defined above), the third eBoard in that SEM must at least have 14 pins. Therefore, such a value-assignment removes 8_PIN from the valid domain of the type of the third eBoard, resulting in the pruned valid domain {16_PIN, 32_PIN, 64_PIN}.

The impacts of the decisions are then reported to the user, in terms of reduced valid domains.

Value inference. After value-assignment propagation and pruning, the tool checks if the size of any valid domains is reduced to one. The configurable parameters with singleton valid domains are set to their only possible value. This enables automatic inferences of values for some configurable parameters, therefore, saving a number of value-assignment steps from the user. For example, in Figure 3 there is a one-to-one deployment relationship between SEM and SemApp. As a result, whenever the user creates a new instance of SEM the tool automatically creates a new instance of SemApp and correctly configures in it the cross-reference to the SEM. Inferring a value for a configurable parameter that represents the size of an object collection, is followed by automatically creating and adding to that collection the required number of objects.

5.2 Constraint satisfaction to provide guidance and automation

The main computation required for providing the aforementioned guidance and automation is the calculation of valid domains through pruning the domains of all the yet-to-be-configured parameters after each configuration iteration using the user's configuration decision.

In our approach, we use a constraint solver over finite domains to calculate the valid domains. In this approach, the configuration space of a product family forms a *constraint system* composed of a set of variables, $x_1, ..., x_n$, and a set of constraints, $\mathcal{C}$, over those variables. Variables represent the configurable parameters, and get their values from the *finite domains* $\mathcal{D}_1, ..., \mathcal{D}_n$. A finite domain is a finite collection of tags, that can be mapped to unique integers. We extract the finite domains of variables from the types of the configurable parameters, enumerations, multiplicities, and OCL constraints in the SimPL model. The constraint set $\mathcal{C}$ includes both the OCL constraints and the information, e.g., multiplicities, extracted from the class diagrams in the SimPL model. A configuration in this scheme corresponds to a (possibly partial) evaluation of the variables $x_1, ..., x_n$. Using a constraint solver the consistency of a configuration w.r.t the constraint set $\mathcal{C}$ is checked, and the valid domains, $D^*{}_1, ..., D^*{}_n$, for all the variables are calculated.

At each value-assignment step during the configuration, a value v_i is assigned to a variable x_i. This value assignment forms a new constraint $c : x_i = v_i$, which is added to the

constraint set $\mathcal{C}$. The added constraint is then propagated throughout the constraint system to identify the impacts of the assigned value on other variables, and to prune and update the valid domains of those variables. This process is realized through a simple and efficient Constraint Programming technique called *constraint propagation* [15]. Constraint propagation is a monotonic and iterative process. During constraint propagation, constraints are used to filter the domains of variables by removing inconsistent values. The algorithm iterates until no more pruning is possible.

Assigning a value to a variable representing the size of a collection relates to adding items to, or removing items from the collection. Adding an item to a collection implies introducing new variables to the constraint system. Similarly, removing items from a collection implies removing variables from the constraint system. As a result, to identify the impacts of changing the size of a collection, new variables have to be added or removed during constraint propagation. This is possible as constraint propagation does not require the set of initial variables to be known a priori. However, the process is no longer monotonic in that case and may iterate forever. In our application, the number of added variables is always bounded, avoiding any non-termination problems.

In our approach, we allow users to modify the previously assigned values as long as the modification does not give rise to any conflict. Since we always keep valid domains of all the configurable parameters up-to-date, conflicts can be detected simply by checking whether the new value is still within the valid domain of the modified configurable parameter. In the following section, we further elaborate on the design of a tool implementing the configuration process presented above.

6 Prototype tool

Figure 4 shows the architecture of the configuration engine that provides the guidance and automation mentioned in Section 5. Inputs to the engine are the generic model of the product family, and the user-provided configuration data. The configuration process starts by loading the generic model. From the loaded model, the configuration engine extracts the first set of the configurable parameters. These configurable parameters are

presented to the user via the interactive user interface for collecting configuration decisions. In addition, the configuration engine generates a constraint model from the input model of the product family. This constraint model is implemented in `clpfd`, a library of the *SICStus Prolog* environment [4, 8]. In `clpfd`, each configurable parameter is represented by a logic variable, to which is associated a finite set of possible values, called a finite domain. After the generic model is loaded, the configuration engineer starts an interactive configuration session for entering configuration decisions.

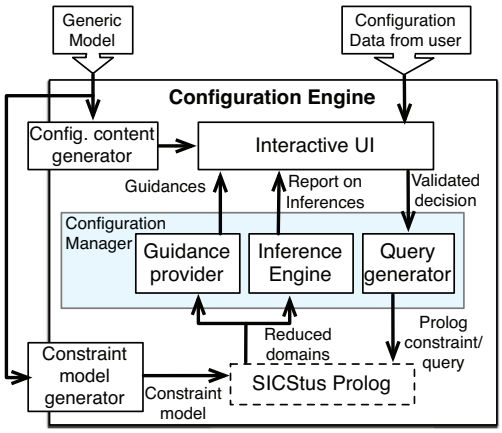


Figure 4: Architecture of the configuration tool.

The configuration engine iteratively and interactively collects configuration decisions from the user. At each iteration, the user enters the values for one or more configurable parameters. Using the domains of the configurable parameters, the consistency of the configuration decisions is checked. If the entered values are all consistent, the *Query generator* is invoked to create a new Prolog query representing a constraint system that contains all the constraints created from the collected configuration decisions. This Prolog query is then used to invoke constraint propagation in order to prune the domains. The new domains serve as inputs to the *Inference engine*, which implements the inference mechanism explained in Section 5.1 to infer values, and the *Guidance provider*, which reports the impacts of configuration choices (e.g., updated domains).

6.1 The `clpfd` library of SICStus Prolog

Choosing Prolog as a host language for developing our configuration engine has several advantages. First, Prolog is a well-established declarative and high-level programming language, allowing fast prototyping for building a proof-of-concept tool, and containing all the necessary interfaces to widely-used programming languages such as Java or C++. In our

tool development, we have used the `jasper` library that allows invoking the SICStus Prolog engine from a Java program. Second, as it embeds a finite domains constraint solver through the `clpfd` library, this allows us to benefit from a very efficient implementation of constraint propagation [9], and all the available constructs (e.g., combinatorial constraints) that have been proposed for handling other applications.

6.2 Mapping to `clpfd`

To use the finite domains constraint engine of SICStus Prolog, we need to translate an ICS product specification into `clpfd`. This requires: (1) translating the SimPL model characterizing the ICS family, and (2) translating the instance model representing the product.

In the first translation, we create a Prolog/`clpfd` program capturing the UML classes, the relationships between the classes, and the OCL constraints of the SimPL model. Our approach for this translation is very similar to a generic UML/OCL to Prolog translation given by [7]. Briefly, we map UML classes and relationships to Prolog compound terms, and every OCL (sub)expression to a Prolog *rule* whose variables correspond to the variables of the given OCL (sub)expression.

In the second translation, given an instance model, we create a SICStus Prolog query to evaluate conformance of the instance model to its related SimPL model (consisting of classes, their relationships, and OCL constraints) captured as a Prolog program as discussed above. To build such query, we map each instance in the given instance model to a Prolog list, and map every configurable parameter of that instance to an element of that list. A configurable parameter that is not yet assigned to a value becomes a variable in the list. For example, a SICStus Prolog query related to an instance model looks like check_product(Als, Ids), where Als is the list representation of all instances, and Ids is the list of the identifiers of instances. The query generator in our tool is responsible for generating these two lists from the instances created and configured by the user. Given the query check_product(Als, Ids), the constraint engine checks whether the instance model specified by Als and Ids conforms to the input SimPL model, and if so, it provides the valid domains for all the variables in Als. Note that the calculation of the valid domains terminates because Als contains a finite number of variables

(as the number of the instances in the product are finite), and all variables take their values from finite domains.

7 Evaluation

To empirically evaluate our approach, we performed several experiments which are reported in this section. The experiments are designed to answer the following three main research questions:

1. What percentage of the value-assignment steps can be saved using our automated configuration approach?

2. How much do the valid domains shrink at each iteration of configuration?

3. How long does it take to propagate a user's decision and provide guidance?

Saving a number of value-assignment steps is expected to reduce the configuration effort, and reduction of the domains decreases the complexity of decision making. Therefore, answers to the first two research questions provide insights on how much configuration effort can be saved. Answering the third research question provides insights into the applicability and scalability of our technique.

To answer these questions we designed an experiment in which we rebuilt three verified configurations from our industry partner using our configuration tool. One configuration belongs to the environmental stress screening (ESS) test of the SEM hardware, which we refer to in this section as the ESS Test. The other two are the verified configurations of two complete products, which we refer to in this section as Product_1 and Product_2. Table 1 summarizes the characteristics of these configurations. We performed our experiments using the simplified generic model of the subsea product family given in Section 4. Number of objects and variables in Table 1 are calculated w.r.t. that simplified model.

Table 1: Characteristics of the rebuilt configurations.

	# XmasTrees	# SEMs	# Devices	# Objects	# Variables
ESS Test	1	1	111	226	343
Product_1	9	18	453	1396	2830
Product_2	14	28	854	2619	5307

We report in Sections 7.1-7.3 the evaluation and analysis that we performed on the experiments to answer the above research questions. At the end of this section, we also discuss some limitations, directions for future work, and the generalizability of our approach.

7.1 Inference percentage

The configuration effort required for creating the configuration of a product is expected to be proportional to the number of configuration iterations and the number of value-assignment steps. Automating the latter is therefore expected to save configuration effort and minimize chances for errors. To measure the effectiveness of our approach in reducing the number of value-assignment steps, we have defined an *inference rate* which is equal to the number of inferred decisions divided by the total number of decisions:

$$inference\ rate = \frac{inferences}{manual\_decisions + inferences} \quad (1)$$

Table 2 shows the inference rates in each case.

Table 2: Inference rates.

	# Manual decisions	# Inferred decisions	Inference rate (%)
ESS Test	373	16	4.11
Product_1	1459	1426	49.42
Product_2	2802	2783	49.82

Note that the inference rate for Product_1 and Product_2 is very close to 50 %. This is because of the *structural symmetry* that exists in the architecture of the system. Structural symmetry is achieved in a product when two or more components of the system have identical or similar configurations. We have modeled the structural symmetries using two OCL constraints.

One specifies that each XmasTree has two SEMs (*twin SEMs*) with identical configurations (i.e., identical number and types of electronic boards and devices connected to them). The other specifies that all the XmasTrees in the system have similar configurations (e.g., all have the same number and types of devices). The first OCL constraint applies to both Product_1 and Product_2, while the second applies to Product_2 only. As a result, the inference rate for Product_2 is slightly higher than that for Product_1. Neither of the OCL constraints applies to the ESS Test, which contains only one XmasTree and one SEM. Therefore, it shows a very low inference rate. In general, the architecture of the product family, and characteristics of the product itself (e.g., structural symmetry) can largely affect the inference rate.

Our experiment shows that our approach can automatically infer a large number of consistent configuration decisions specially for products with some degree of structural symmetry. Assuming automated value-assignments have similar complexity to manual ones, our approach can save about 50% of the configuration effort of Product_1 and Product_2.

7.2 Reduction of valid domains

Pruned domains are the output of constraint propagation. Pruning of the domains decreases the complexity of decisions to be made. As part of our experiment, we measured how the domains shrink after each constraint propagation step. Such reduction of the domains is measured by comparing the size of each pruned domain before and after constraint propagation. This is possible and meaningful because all the domains are finite. Table 3 shows the average reduction of domains for each case. *Reduction rate* in the table is defined as the proportion of the *reduction size* (i.e., number of distinct values removed from a domain) to the initial size of the domain (i.e., the number of distinct values in a domain). In the calculations in Table 3 we have not considered domain reductions that resulted in inferences. This result shows that the domains of variables can be considerably reduced when a value is assigned to a dependent variable. Specifically, it shows that, on average, after each value-assignment step 37.98% of the values of the dependent variables are invalidated. Without such a dynamic recomputation of valid domains, there would be a higher risk for the user to make inconsistent configuration decisions. Moreover, comparing the inference rate from Table 2 and the reduction rate from

Table 3 over the three cases suggests that while structural symmetry can highly affect the inference rate, it does not have a large impact on the reduction rate.

Table 3: Average shrinking of the domains.

	Count*	**Avg. initial domain size**	**Avg. reduction size**	**Avg. reduction rate**
ESS Test	732	30.557	13.803	45.17 %
Product_1	2564	62.125	21.367	34.39 %
Product_2	7557	35.97	14.205	39.49 %
			Avg. over all cases:	**37.98 %**
* total number of domains that have been pruned or reduced. Avg.: the average over all reduced domains in the whole configuration.				

7.3 Constraint propagation efficiency

Providing automation and guidance as part of the interactive configuration process requires the underlying computation to be sufficiently efficient for our approach to be practical.

We define the efficiency of our approach as the amount of time needed for validating and propagating the user decision. For this purpose, we have measured at each constraint propagation step the execution time, and the number of variables in the constraint system. Figure 5 shows the average time required for propagating user decisions after each value-assignment step. As shown in this figure, for products with less than 1000 variables, it takes, on average, less than one second to validate and propagate the decision. However, this time grows polynomially with the number of variables, which itself is proportional to the number of instances.

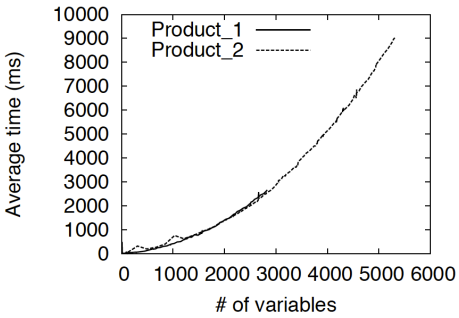


Figure 5: Constraint propagation time grows quadratically with the number of variables (with a coefficient of determination of 0.9994).

Since in our experiment we have used a simplified model of the product family, we expect that for a complete model of the system the number of instances and the number of variables be much higher than that in this experiment. However, our experiment shows that not all of these variables are dependent on each other. To provide an insight into the level of

dependency between variables, for each case, we can compute the average number of reduced domains. The average number of reduced domains is 1.8 (2564 from Table 3 divided by 1459 from Table 2) for Product_1 and, 2.7 for Product_2. In other words, on average, each variable in Product_1 (Product_2) is dependent to less than two (three) other variables. The polynomial ($O(n^2)$) growth of the execution time is, however, due to our current implementation, in which, we compute the valid domains of all variables (not only the dependent variables) by creating a new constraint propagation session after each value-assignment step. Therefore, we expect that by optimizing our implementation and incrementally adding new constraints to an existing constraint propagation session we can significantly improve the efficiency of our approach. Such an optimization requires an additional preprocessing step before creating queries and invoking the constraint solver. This needs to be investigated in more depth and is left for future research.

7.4 Discussion

Limitations and directions for future work. The inference rate and the reduction rate, in addition to be affected by the architecture of the product family, are affected by the order in which the decisions are made. An *optimal order* of applying configuration decisions can be defined as the order which can result in the maximum inference rate and reduction rate. The optimal order can be reported to the user as additional guidance. Our current implementation does not provide such a guidance and therefore the results reported in this paper are probably, a lower bound for potential configuration effort savings. It is therefore important that in the future we support the optimization of the ordering to maximize inferred decisions and the reduction of domains. Devising criteria and heuristics for finding such optimal order is one direction of our future work.

Another research question is "How useful is the guidance provided by our approach?". Answering this question requires conducting an experiment involving human subjects. This experiment is also part of future work.

Generalizability of our approach. Like any other model-based engineering approach, the effectiveness of our approach depends on the quality of the input generic models. Our configuration approach can be used to configure only the variabilities that are captured in

the generic model of the product family. Similarly, the approach can validate the decisions and automatically infer decisions only based on the dependencies that are captured in the model. Our evaluation in this paper shows that the SimPL methodology and notations that we proposed in [5] enables the creation of models of the required quality.

The use of a constraint solver over finite domains limits our approach to the constraints that capture restrictions on variables with finite domains. Constraint solvers over continuous domains are available to overcome this limitation but their integration with an efficient finite domains solver is still an open research problem [10]. Moreover, as we have not encountered this type of constraint with our industry partner, we don't expect this to be a restriction in our context.

8 Related Work

Most of the existing work on constraint-based automated configuration in product-line engineering focuses on resolving variabilities specified by feature models [17] and their extensions [11]. Basic feature models cannot express complex variabilities or dependencies required for configuring embedded systems [5]. However, extended feature models that allow attributes, cardinalities, references to other features, and cloning of features are, as mentioned in [12], as expressive as UML class diagrams and can be augmented by OCL or XPath queries to describe complicated feature relationships as well.

We compare our work with the existing automated configuration and verification tools proposed for extended feature models since these are the closest to our SimPL models. FMP [12] is an Eclipse plug-in that enables creation and configuration of extended feature models. FMP can verify full or partial configurations for a subset of extended feature models, specifically those with boolean variables and without clonable features. FAMA [6] drops this limitation and can verify extended feature models with variables over finite domains. However, FAMA is more targeted towards the verification and analysis of feature models. Therefore, it does not handle validating partial configurations or help build full configurations iteratively. Finally, Mazo et. al. [19] use constraint solvers over finite domains to analyze extended feature models. This approach is the closest to ours as it can handle all the advanced

constructs in extended feature models, and further enables verification of full and partial configurations.

The main limitation of all of the above approaches is that none of them supports verification and analysis of complex constraints such as those in Section 4.1. These constraints express complex relationships between individual elements or collections of elements and are instrumental in describing software/hardware dependencies and consistency rules in embedded systems. Our tool, in addition to verifying these constraints, provides interactive guidance to help engineers effectively build configurations satisfying these constraints. Finally, to the best of our knowledge, none of the above approaches have been applied to nor evaluated on real case studies.

More recently, constraint satisfaction techniques have been used to automate configuration in the presence of design or resource constraints [14, 16]. The main objective is to search through the configuration space in order to find optimized configurations satisfying certain constraints. Our work, however, focuses on inter- actively guiding engineers to build consistent product configurations, a problem that we have shown earlier in our paper to be important in practice. We do not intend to replace human decision making during configuration. Instead, we plan to support engineers when applying their decisions in order to reduce human errors and configuration effort.

In contrast to related work in [14, 16], we enable users to interact with the constraint solver during the search. This is because supporting user guidance and interactive configuration are paramount to our approach. As a result, we require a technique that is fast enough for instant interaction with users and therefore cannot rely on dynamic constraint solving, which the authors in [16] have shown to be orders of magnitude slower than the SICStus CLP(FD) library. As for DesertFD in [14], it neither provides user guidance nor enables interactive configuration.

9 Conclusion

In this paper, we presented an automated model-based configuration approach for embedded software systems. Our approach builds on generic models created in our earlier work, i.e.,

the SimPL models, and uses constraint solvers to interactively guide engineers in building and verifying full or partial configurations. We evaluated our approach by applying it to a real subsea production system where we rebuilt three verified configurations of this system to evaluate three important practical factors: (1) reducing configuration effort, (2) reducing possibility of human errors, and (3) scalability. Our evaluation showed that, in our three example configurations, our approach (1) can automatically infer up to 50% of the configuration decisions, (2) can reduce the size of the valid domains of the configurable parameters by 40%, and (3) can evaluate each configuration decision in less than 9 seconds.

While our preliminary evaluations demonstrate the effectiveness of our approach, the value of our tool is likely to depend on its scalability to very large and complex configurable systems. In particular, being an interactive tool, its usability and adoption will very much depend on how fast it can provide the guidance information at each iteration. Our current analysis shows that the propagation time grows polynomially with the size of the product. But we noticed in our work that after each iteration only a very small subset of variables are affected. Therefore, if we could reuse the analysis results from the previous iterations, we could possibly improve the time it takes to analyze each round significantly.

Bibliography

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

[2] MARTE: Modeling and Analysis of Real-Time and Embedded Systems. http://www.omgmarte.org/, 2012.

[3] OCL: Object Constraint Language. http://www.omg.org/spec/OCL/2.2/, 2012.

[4] SICStus Prolog. `www.sics.se/sicstus/`, February 2012.

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

[6] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz Cortés. FAMA: tooling a framework for the automated analysis of feature models. In *VaMoS*, 2007.

[7] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW*, 2008.

[8] M. Carlsson and P. Mildner. SICStus Prolog – the first 25 years. *CoRR*, abs/1011.5640, 2010.

[9] M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP*, 1997.

[10] H. Collavizza, M. Rueher, and P. V. Hentenryck. A constraint-programming framework for bounded program verification. *Constraints Journal*, 2010.

[11] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 2005.

[12] K. Czarnecki and P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Workshop on Software Factories at OOPSLA*, 2005.

[13] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *GPCE '06*, 2006.

[14] B. K. Eames, S. Neema, and R. Saraswat. DesertFD: a finite-domain constraint based tool for design space exploration. *Design Autom. for Emb. Sys.*, 14(2), 2010.

[15] P. V. Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(FD). In *Selected Papers from Constraint Programming: Basics and Trends*, 1995.

[16] Á. Horváth and D. Varró. Dynamic constraint satisfaction problems over models. *Software and Systems Modeling*, 2010.

[17] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, 1990.

[18] R. E. Lopez-Herrejon and A. Egyed. Detecting inconsistencies in multi-view models with variability. In *ECMFA*, 2010.

[19] R. Mazo, C. Salinesi, D. Diaz, and A. Lora-Michiels. Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In *ENASE*, 2011.

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

Paper 4:
A Modeling Approach to Support the Similarity-Based Reuse of Configuration Data

A Modeling Approach to Support the Similarity-Based Reuse of Configuration Data

Razieh Behjati[1], **Tao Yue**[1], **Lionel Briand**[1,2,3]

[1] Certus Verification and Validation Center, Simula Research Laboratory,
P. O. Box 134, N-1325 Lysaker, Norway

[2] Department of Informatics, University of Oslo,
P. O. Box 1080 Blindern, N-0316 Oslo, Norway

[3] SnT Centre, University of Luxembourg
Luxembourg

Abstract:

Product configuration in families of Integrated Control Systems (ICSs) involves resolving thousands of configurable parameters and is, therefore, time-consuming and error-prone. Typically, these systems consist of highly similar components that need to be configured similarly. For large-scale systems, a considerable portion of the configuration data can be reused, based on such similarities, during the configuration of each individual product. In this paper, we propose a model-based approach to automate the reuse of configuration data based on the similarities within an ICS product. Our approach enables configuration engineers to manipulate the reuse of configuration data, and ensures the consistency of the reused data. Evaluation of the approach, using a number of configured products from an industry partner, shows that more than 60% of configuration data can be automatically reused using our similarity-based approach, thereby reducing configuration effort.

1 Introduction

Modern society is increasingly dependent on embedded software systems such as Integrated Control Systems (ICSs). Examples of ICSs include industrial robots, process plants, and oil and gas production platforms. Many ICS producers apply product-line engineering to develop the software embedded in their systems. They typically build a generic software, specifying a large number of interdependent configurable parameters, that need to be configured for each product according to the product's hardware architecture [5]. To configure the generic software, engineers manually assign values to tens of thousands of configurable parameters, while accounting for the constraints and dependencies between them. This makes software configuration time-consuming, error-prone, and challenging.

In the literature, the area of product configuration is still rather immature [22] and largely concentrates only on resolving high-level variabilities in feature models [19] and their extensions [10, 11]. Feature models, however, are not easily amenable to capturing complex architectural variabilities and dependencies in embedded systems. Consequently, existing configuration approaches do not focus on configuration challenges in highly-configurable embedded systems, where large numbers of configurable components need to be configured and cloned.

In a previous study [5], we identified characteristics of ICS families, and their configuration challenges. Our studies show that ICSs, like many other embedded systems, bear a high degree of structural similarity within their hardware architectures to fulfill several product requirements, related for example to the environment, safety, and cost efficiency. Structural similarities in hardware affect software design and configuration, i.e., similar patterns of configuration are repeated throughout the software configuration.

In this paper, we devise a model-based approach to automatically infer configuration decisions based on the internal structural similarities of a product and previously made decisions. Our solution (1) includes a similarity modeling approach for capturing structural similarities in terms of architectural elements in an ICS family model, (2) applies feature models in practice to provide user-level representations of structural similarities so as to enable controlling the required amount of configuration reuse through feature selection, and (3) enables reducing

configuration effort in large-scale, highly-configurable ICSs based on structural similarities. We build on our previous work, where we proposed a modeling methodology [5, 6], called SimPL, for modeling families of ICSs, and a model-based configuration approach [4] that uses finite domains constraint solving to automate and interactively guide consistent configuration of such systems.

We motivate the work and formulate the problem in Section 2, by explaining the current practice in configuration reuse. We analyze the related work in Section 3. An overview of our model-based solution is given in Section 4. An example ICS family illustrating the main aspects of the SimPL methodology is presented in Section 5. We explain our similarity modeling approach in Section 6. The use of feature selection to control configuration reuse, and constraint propagation to automate configuration reuse are presented in Sections 7 and 8. We evaluate the effectiveness of our approach in Section 9. Finally, we conclude the paper in Section 10.

2 Configuration reuse: practice and problem definition

Figure 1 shows a simplified model of a fragment of a subsea production system produced by our industry partner. As shown in the figure, products are composed of mechanical, electrical, and software components. Our industry partner, similar to most companies producing ICSs, has a generic product that is configured to meet the needs of different customers. For example, different customers may require products with different numbers of subsea Xmas trees. A Xmas tree in a subsea production system provides mechanical, electrical, and software components for controlling and monitoring a subsea well.

Configuration in the ICSs domain is typically performed in a top-down manner where the configuration engineer starts from the higher-level components and determines the type and the number of their constituent (sub)components. Some components are invariant across different products, and some have parameters (i.e., *configurable parameters*) whose values differ from one product to another. The latter group, known as *configurable components*, may need to be further decomposed and configured. In the rest of this paper, whenever clear from the context, we use *configuration* to refer either to the configuration process or to the

description of a configured artifact.

Subsea production systems, and in general ICSs, are typically large-scale systems with thousands of components and tens of thousands of configurable parameters. Usually, in these systems, a high degree of similarity is required among different configurable components to fulfill certain product requirements such as environmental, safety, or cost efficiency. For example, to reduce the costs of design and production, it may be required that all the Xmas trees in a product contain the same number and types of devices, thus requiring all the controller software units (SemApplications) to be configured similarly.

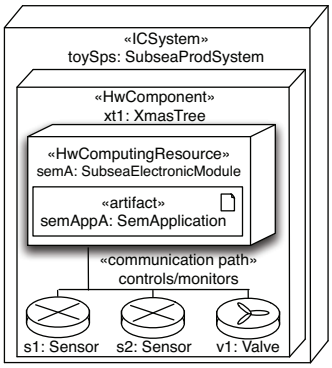


Figure 1: Fragment of a simplified subsea production system.

Similarity, in this context, is defined as a relationship between two or more configurable components. Two configurable components are similar if a subset of their configurable parameters have identical values. Such configurable components are not themselves identical, as some of their configurable parameters may have different values. The similarity that exists in such systems enables the reuse of configuration data: instead of configuring every configurable parameter separately, configurable parameters with identical values can be configured all at once. The large number of configurable parameters and the high degree of similarity lead to the potential for a high degree of configuration reuse. This can considerably reduce the *configuration effort*, which we define to be proportional to the number of manual configuration decisions.

Configuration is currently done in our industry partner using an in-house tool with primitive support for configuration reuse through a copy and paste mechanism. The existing support for the reuse of configuration data at our industry partner has the following limitations: (1) It does not provide the user with sufficient control over the configuration reuse. The user can only select one subcomponent and duplicate its whole configuration. As a result, it is sometimes necessary to modify the values of some configurable parameters in the duplicated subcomponents. (2) It does not automatically enforce the reuse of configuration data. The configuration engineer has to derive, based on her own knowledge and experience, a

configuration reuse plan that specifies what data should be reused and how. The configuration tool cannot help following the configuration reuse plan. (3) Changes in the configuration data are not automatically propagated to the copies, therefore resulting in inconsistencies.

In our previous work [4, 6], we proposed a model-based configuration approach that ensures the consistency of a, possibly partial, product during the configuration process. In this paper, we build on our previous work to propose an approach for modeling structural similarities in ICSs to automatically reuse configuration data while preventing all the above-mentioned limitations.

3 Related work

Feature models [10, 19] have been most commonly studied in the literature (e.g., [9, 16, 20]) for specification and model-based analysis of product families. However, few industrial applications (i.e., [13, 15, 23, 25]) of feature models have been reported according to the findings of a preliminary review presented in [18]. Another group of approaches, which address architecture-level variability modeling (e.g., [17, 21, 24, 27]), are studied and evaluated in our previous work [5, 6]. Structural similarities within individual products, and modeling solutions to capture them are, however, missing from these approaches and applications.

Practical challenges in the configuration of highly-configurable systems have been studied, and large numbers of configurable parameters and their implicit interdependencies have been categorized as one major source of configuration errors [12]. Moreover, results from a systematic literature review [22] confirm that automation is one of the most important requirements for configuration and product derivation support. Related work on automated verification and guidance during configuration is presented in our previous work [4]. To the best of our knowledge, however, there is no work in the literature focusing on the automated reuse of configuration data, or on the similarity-based approaches to improve or automate configuration. In this paper, we address this gap by proposing a model-based approach to the automated reuse of configuration data based on structural similarities in large-scale, highly-configurable embedded systems.

4 Overview of our approach

Figure 2 shows an overview of our *reuse-oriented* configuration approach, which is a model-based approach to the automated reuse of configuration data based on the similarities that exist within a particular product. This approach is an extension to our previous work (the upper part in Figure 2) on automated, model-based configuration, which has two major steps. In the first step, we build a configurable and generic model for an ICS family (the Product-family modeling step). In the second step, the Guided configuration step, we interactively guide users to generate specifications of particular products complying with the generic model built in the first step.

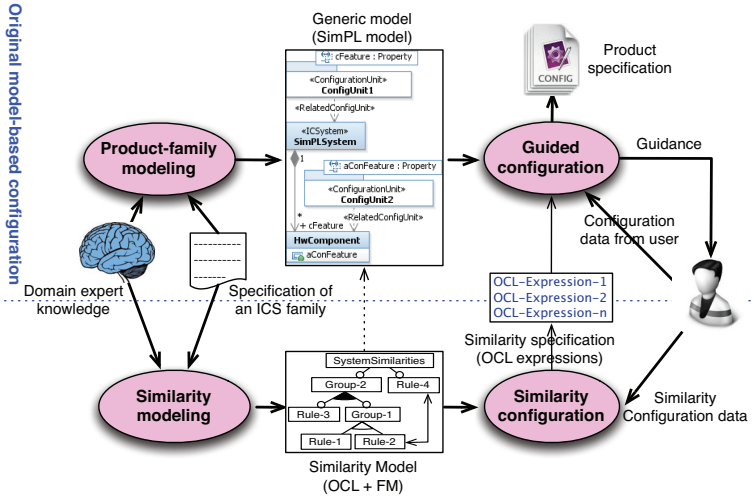


Figure 2: An overview of our reuse-oriented configuration approach.

As shown in the lower part of Figure 2, in our reuse-oriented configuration approach, we have extended both the modeling step and the configuration step of the original configuration approach. Therefore, the reuse-oriented configuration approach has four major steps. In the first step, the Product-family modeling step, a configurable and generic model of an ICS family is created by following the SimPL methodology [5, 6]. In the second step, the Similarity modeling step, possible structural similarities that may exist in some particular products are modeled and organized in a *similarity model*. In the third step, the Similarity configuration step, the similarity model is used to generate *similarity specifications* of particular products. Finally,

in the Guided configuration step, we use our existing automated configuration approach [4] to interactively guide users to generate specifications of particular products that comply both with the generic SimPL model of the product family and with the similarity specifications of the products generated in the previous step.

Step 1: Product-family modeling

During the product-family modeling step, we provide domain experts with a modeling methodology, called SimPL [5, 6], to manually create a product-family model describing an ICS family. The SimPL methodology enables the domain experts to create, from textual specifications and tacit domain knowledge, architecture models of ICS families that encompass, among other things, information about variabilities and consistency rules. We briefly describe and illustrate the SimPL methodology in Section 5. Note that our reuse-oriented extension has no impact on the product-family modeling step. This step is performed exactly as it is done in our original configuration approach.

Step 2: Similarity modeling

During the similarity modeling step, domain experts follow the similarity modeling approach presented in this paper to manually create similarity models from textual specifications and their own domain knowledge. A similarity model expresses the structural similarities in two levels of abstraction. In the lower level of abstraction, OCL is used to express the similarity in terms of the model elements in the SimPL model of the product family. Each OCL constraint in this level specifies one *similarity rule*. In the higher level of abstraction, a feature model [19] is used to provide a user-level representation of the similarity rules. This feature model captures the variability that exists among individual products with respect to the applicability of the similarity rules. We describe and illustrate our approach to similarity modeling in Section 6.

Step 3: Similarity configuration

During the similarity configuration step, configuration engineers use the feature models created in the previous step to select, for each product, the applicable similarity rules according to the needs of that particular product. The result of this step is a similarity specification, which is a collection of OCL constraints each representing one applicable similarity rule. Using feature models as the user-level representation of similarity rules, configuration engineers can generate similarity specifications without requiring to know OCL or the SimPL methodology. In addition, by organizing the similarity rules (that can result in the reuse of configuration data) and their variabilities in a feature model, we provide configuration engineers with a suitable mechanism to gain control over the reuse of configuration data. This way, we address the first limitation of the existing support for configuration reuse as discussed in Section 2. Similarity configuration is illustrated in Section 7.

Step 4: Guided configuration

During the guided configuration step, configuration engineers create full or partial product specifications by resolving variabilities in a product-family model. Inputs to the guided configuration step are the generic model of the product family and the similarity specification of the product. We use these two inputs to ensure the consistency of the product specification during the entire configuration process. For this purpose, we use a finite domains constraint solver to validate each user decision, and to identify the impacts of each decision. As an impact of a user decision, the constraint solver may infer the values of one or more configurable parameters. We refer to this as the reuse of configuration data.

The main idea in this work is to use the similarity rules in the similarity specifications to trigger the inference capability of the constraint solver to automatically enforce the reuse of configuration data. Moreover, to keep the product specification consistent with respect to the similarity rules, whenever the value of a configurable parameter is changed the new value is automatically propagated to replace the related inferred values. Therefore, using our extended

configuration approach, we address the second and third limitations discussed in Section 2. Note that, in this work, we have extended our original guided configuration step only by adding to it one extra input, which is the similarity specification. However, this simple extension automatically results in the automated similarity-based reuse of configuration data. This is described in details together with a brief description of our original guided configuration step in Section 8. Our original guided configuration step is described in details in [4].

5 A subsea product-family model

The SimPL methodology organizes a product-family model into two views: a *system design view*, and a *variability view*. The system design view presents both hardware and software entities of the system and their relationships using UML classes [1]. The variability view, on the other hand, captures the set of system variabilities using a collection of *configuration units*. Each configuration unit is related to exactly one class in the system design view and defines a number of *configurable features*. Each configurable feature describes a variability in the value, type, or cardinality of a property in the corresponding class. In addition to the two views described above, each SimPL model has a repository of OCL expressions [2]. These OCL expressions specify constraints among the values, types, or cardinalities of different properties of different classes. We call these OCL constraints *universal consistency rules*, as they are part of the product-family commonalities and must hold for all the products in the family.

Figure 3 shows a fragment of the SimPL model for a simplified subsea production system[16], SubseaProdSystem. In a subsea production system, the main computation resources are the Subsea Electronic Modules (SEMs), which provide electronics, execution platforms, and the software required for controlling subsea devices. SEMs and Devices are contained by XmasTrees. Devices controlled by each SEM are connected to the electronic boards of that SEM. Software deployed on a SEM, referred to as SemAPP, is responsible for controlling and monitoring the devices connected to that SEM. SemAPP is composed of a number of DeviceControllers, which is a software class responsible for communicating with, and

[16]This example is a sanitized fragment of a subsea production case study [5].

controlling or monitoring a particular device. The system design view in Figure 3 represents the elements and the relationships discussed above.

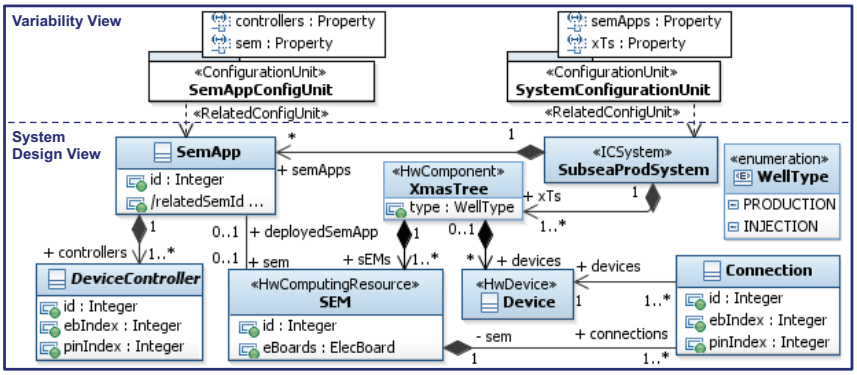


Figure 3: A fragment of the SimPL model for the subsea production system.

The variability view in the SimPL methodology is a collection of template packages, each representing one configuration unit. The upper part in Figure 3 shows a fragment of the variability view for the subsea production system. Due to the lack of space we have shown only two template packages in the figure. As shown in the figure, the package SystemConfigurationUnit represents the configuration unit related to the class SubseaProdSystem in the system design view. Template parameters of this package specify the configurable features of the subsea production system, which are: the number of XmasTrees, and SEM applications (semApps).

A number of universal consistency rules are defined for the subsea production system in Figure 3. Below are OCL expressions for two of these consistency rules.

```
context Connection inv PinRange
self.pinIndex >= 0 and self.sem.eBoards->asSequence()->
     at(self.ebIndex+1).numOfPins > self.pinIndex
context Connection inv BoardIndRange
self.ebIndex >= 0 and self.ebIndex < self.sem.eBoards->size()
```

The first constraint states that the value of the pinIndex of each device-to-SEM connection must be valid, i.e., the pinIndex of a connection between a device and a SEM cannot

exceed the number of pins of the electronic board through which the device is connected to its SEM. The second constraint specifies the valid range for the ebIndex of each device-to-SEM connection, i.e., the ebIndex of a connection between a device and a SEM cannot exceed the number of the electronic boards on its SEM.

Product specifications are created from family models by instantiating the classes associated to configuration units, and assigning values to the configurable parameters (i.e., instances of configurable features) of those instances.

6 Similarity modeling

As mentioned in Section 4, in the similarity modeling step, we create similarity models that specify the similarity rules in two levels of abstraction. In this section, we first define and exemplify[17] the similarity rules. Then we explain how OCL can be used to model similarity rules in terms of the model elements in the SimPL model of the product family. Then we explain how feature models are used to provide a user-level representation of similarity rules and their variabilities. Finally, we explain the refactoring of similarity models.

6.1 Similarity rules

A similarity rule specifies a relationship between two or more configuration unit instances within a particular product. Two configuration unit instances are similar if a subset of their configurable parameters have equal or identical values. For example, a similarity rule named XtTypeSimilarity specifies that all the Xmas trees (Figure 3) in a subsea product must be of the same type. Here, Xmas trees are the configuration units that are required to be similar. Types of the Xmas trees, which can either be production or injection, are the configurable parameters that are required to be identical for the similarity rule to hold.

Every similarity rule has two parts: a *scope*, and a *similarity relation*. The scope of a similarity rule determines the configuration unit instances that must be similar. For example,

[17] Examples in this section focus on describing hardware similarities, as the SimPL model in Figure 3 mostly contains hardware classes. However, in practice, similarity rules are mainly defined in terms of software classes, as they are intended to be used for reusing software configuration decisions. Note that, software similarities in a product family are, in general, very similar to its hardware similarities.

the scope of the similarity rule XtTypeSimilarity is the set of all Xmas trees in the product. The similarity relation in a similarity rule specifies how the similarity is achieved. It is normally composed of one or more equality relationships. Each relationship relates the values of different instances of a particular configurable feature, each belonging to a configuration unit instance in the scope of the similarity rule. For example, in XtTypeSimilarity, the similarity relation is composed of a single equality relationship that relates the values of the configurable parameter type of all the Xmas trees in the product.

It is possible to have several similarity rules with the same scope, but expressing different aspects of similarity. For example, in addition to XtTypeSimilarity, we can have another similarity rule among all the Xmas trees in the product, named XtSemNumSimilarity, expressing that all of the Xmas trees must have the same number of SEMs.

6.2 Architecture level modeling of similarity rules using OCL

Configuration in our automated, model-based approach is performed by resolving variabilities through assigning values to configurable parameters [4]. To enable the reuse of such configuration decisions based on the similarities within a product, we express the similarity rules in terms of the configurable features and other model elements in the SimPL model of a product family. For this purpose, we use OCL, as it is the standard language for expressing constraints on the elements in UML class diagrams.

Each OCL expression is written in the context of an instance of a specific type [2]. In an OCL expression representing a similarity rule, the context must be the instance that contains all the configuration unit instances that form the scope of the similarity rule. For example, to model the similarity rule XtTypeSimilarity, we use an OCL invariant written in the context of the class SubseaProdSystem. This class is the topmost class in the SimPL model (Figure 3), and contains all the instances of XmasTree[18]. Each equality relationship in the similarity relation of a similarity rule becomes a boolean subexpression in the corresponding OCL invariant. The following is the OCL invariant expressing XtTypeSimilarity.

[18] In the SimPL methodology, each product contains only one instance of the topmost class [5, 6]. In a product specification created from the SimPL model in Figure 3, the only instance of the class SubseaProdSystem contains all the XmasTree instances.

```
context SubseaProdSystem inv XtTypeSimilarityInv
self.xTs->forAll(x | x.type = WellType::PRODUCTION) or
self.xTs->forAll(x | x.type = WellType::INJECTION )
```

The scope of a similarity rule does not always contain all the instances of a configuration unit. In general, for modeling the scope of a similarity rule more expressive OCL constructs such as *implication*- or *selection*-statements are required. The following is an example. This similarity rule specifies that all the production Xmas trees must have two SEM instances. Here, the scope of the similarity rule is the set of all Xmas trees that are of type production (specified using the selection-statement), and the number of SEMs is the configurable feature that must have the same value for all such Xmas trees.

```
context SubseaProdSystem inv ProductionXtTwoSemSimilarityInv
self.xTs->select(x | x.type = WellType::PRODUCTION)
          ->forAll(x | x.sEMs->size() = 2)
```

We use OCL *and*-statements to specify similarity relations that are composed of two or more equality relationships. SemDesignSimilarityInv is an example.

```
context SubseaProdSystem inv SemDesignSimilarityInv
SEM.allInstances()->forAll(s, t | s.eBoards->size() = t.eBoards->size())
and
SEM.allInstances()->forAll(s, t |
      s.eBoards->forAll(e1 | t.eBoards->exists(e2 | e2 = e1)))
```

6.3 User-level modeling of similarity rules using feature models

As mentioned in Section 4, we use feature models [19] to provide a user-level representation of the similarity rules. We call these feature models *similarity feature models*. A similarity feature model captures the variabilities that exist among individual products with respect to the applicability of the similarity rules. A similarity feature model is part of a product-family specification, and is created only once for that product family.

Figure 4 shows a fragment of the similarity feature model for the product family shown in Figure 3. To create a similarity feature model, we follow the existing feature modeling methodologies [3] and organize features into a tree. Each leaf feature in the tree represents a similarity rule and is associated with an OCL expression. For example, XtTypeSimilarity is a leaf feature associated with the OCL invariant XtTypeSimilarityInv.

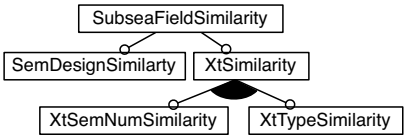


Figure 4: A fragment of the similarity feature model for the subsea production systems family.

Non-leaf features (e.g., XtSimilarity) are used to group related similarity rules, or other non-leaf features. In Figure 4, XtSimilarity is a non-leaf or-feature that groups two leaf features XtTypeSimilarity and XtSemNumSimilarity. An or-feature specifies that one or more of its subfeatures can be selected. Both XtTypeSimilarity and XtSemNumSimilarity are optional features and therefore introduce variabilities that should be resolved during similarity configuration.

Different types of dependencies, such as *imply* and *exclude*, may exist among similarity rules. Using feature models to organize similarity rules allows modeling these dependencies among the features representing the similarity rules. This makes OCL constraints simpler and independent from each other, thus easier to maintain. In general, all similarity rules must be consistent with the universal consistency rules in the SimPL model (This consistency can be checked, for example, using the approaches in [8] and [14]). Similarity rules are, in fact, complementary to the universal consistency rules, but must not be contradictory to them. However, similarity rules can be contradictory to each other. If two similarity rules are contradictory, an exclude or alternative relationship is necessary between the features representing them to avoid any inconsistency in the products. Figure 5 shows an example. The similarity feature model in this figure is achieved by refactoring (Section 6.4) the similarity feature model in Figure 4. AllInjection (AllProduction) is a similarity rule that specifies that all Xmas trees must be of type injection (production). The OCL constraints associated with AllInjection and AllProduction are contradictory and cannot be true simultaneously. To ensure that these two similarity rules are never se-

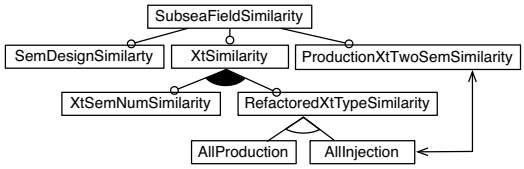


Figure 5: Dependencies between similarity rules are modeled as dependencies between features.

lected simultaneously, the features representing them are grouped in an alternative-feature (RefactoredXtTypeSimilarity). In addition, the similarity feature model in Figure 5 shows an exclude relationship between the features AllInjection and ProductionXtTwoSemSimilarity, as selecting AllInjection makes ProductionXtTwoSemSimilarity void.

6.4 Refactoring similarity models

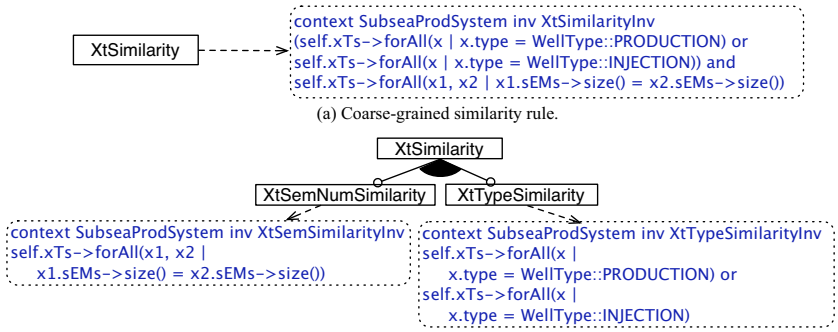
Creating similarity models is an incremental process, which may involve refactoring course-grained similarity rules into more fine-grained ones. Refactoring a similarity rule is done in both the architecture (i.e., OCL expressions) and the feature levels. Refactoring similarity models is, in particular, useful when product families evolve [7, 26] and new requirements are introduced.

Consider the OCL invariant XtSimilarity in Figure 6-(a). XtSimilarity represents a similarity rule that requires all the Xmas trees in the susbea field to be of the same type (i.e., all production or all injection), and that all the Xmas trees have the same number of SEMs. This rule is associated with a single feature in the similarity feature model.

Figure 6-(b) shows the similarity feature model and OCL constraints resulting from refactoring XtSimilarity. This refactoring is done to fulfill the needs of a new product that requires all the Xmas trees in the field to have the same number of SEMs, but does not require all the Xmas trees to be of the same type. The refactoring shown in Figure 6 has decomposed XtSimilarity into two finer-grained similarity rules that can be selected independently during similarity configuration. To fulfill the needs of the new product, one must select the features XtSimilarity and XtSemNumSimilarity and leave XtTypeSimilarity unselected.

In general, if the OCL constraint expressing a similarity rule is a conjunction of subexpressions each expressing an equality relation on a different configurable feature, then it is a good modeling practice to refactor the similarity model by decomposing that similarity rule so that each subexpression becomes an independent similarity rule. To reflect this refactoring step in the similarity feature model, we make the feature corresponding to the original similarity rule a non-leaf or-feature and add to that a number of optional subfeatures each associated with one of the OCL subexpressions. In Figure 6-(b), the two OCL expressions associated

with features XtTypeSimilarity and XtSemNumSimilarity are in fact the two subexpressions of the OCL constraint in Figure 6-(a).



(a) Coarse-grained similarity rule.

(b) Refactored finer-grained similarity rules.

Figure 6: Refactoring of a similarity rule.

As shown in Figure 5, XtTypeSimilarity can be refactored by decomposing its associated OCL constraint into two finer-grained OCL constraints, one (i.e., AllProduction) expressing that all the Xmas trees must be of type production, the other (i.e., AllInjection) expressing that all Xmas trees must be of type injection. This refactoring allows configuration engineers to identify the type of the Xmas trees during the similarity configuration; while, without this refactoring, configuration engineers must make this choice during the guided configuration step. Note that in both cases the total number of configuration decisions to be made are equal. Whether refactoring XtTypeSimilarity or not depends on the requirements of the product family (e.g., presence of ProductionXtTwoSemSimilarity).

7 Similarity configuration

Optional features in the similarity feature model represent variability points that should be resolved during the similarity configuration step to generate similarity specifications. Configuration engineers resolve these variabilities by selecting features in the similarity feature model according to the needs of a particular product. For example, Figure 7 shows the similarity feature model in Figure 5 configured for a product that requires all the Xmas trees to have the same number of SEMs.

Features that are selected during similarity configuration represent the similarity rules that must hold within the product under configuration. OCL constraints associated to the selected features are used to automatically generate the similarity specification of the product. For example, the similarity specification for the product mentioned above, will contain one OCL constraint, which is XtSemNumSimilarityInv that is the OCL constraint associated with XtSemNumSimilarity as shown in Figure 6.

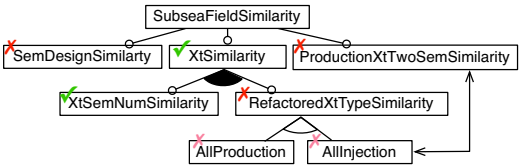


Figure 7: Similarity feature model configured for a particular product.

8 Configuration reuse through constraint propagation

Our original model-based configuration approach, presented in details in [4], gets as input a SimPL model, which is composed of a set of UML class diagrams and a set of OCL constraints. From these inputs, it creates a *constraints system* and uses a finite domains constraint solver to validate user decisions, to ensure the consistency of the configured product, and to automatically infer values.

Originally, OCL constraints that are fed to the configuration engine specify universal consistency rules. As mentioned in Section 4, we extend our original approach by adding to it one more input: the similarity specification of a product. In the reuse-oriented configuration approach, OCL constraints in the similarity specification are merged with the OCL constraints of the universal consistency rules, and are used by the configuration engine to create the constraints system.

Bringing the similarity rules – which express equality relationships among configurable parameters – in the constraints system forces the configuration engine to infer new values whenever a value is assigned to a configurable parameter involved in a similarity rule. For example, as a result of selecting XtTypeSimilarity, when the configuration engineer sets the type of one Xmas tree to production, the type of all other Xmas trees will be automatically set to production.

In general, OCL constraints representing similarity rules are expected to result in high numbers of inferences and a high ratio of reuse of configuration data. Using the similarity feature model and by configuring it (through selecting features), configuration engineers can control the degree of configuration reuse for each product. Note that some of the universal consistency rules may, as well, result in the reuse of configuration data. Table 1 compares universal consistency rules and similarity rules.

Table 1: A comparison between universal consistency rules and similarity rules.

	Applies to	**Modeled in**	**Specifies**	**Impact on reuse**
Universal consistency rule	All products	OCL	All types of relationships	May result in reuse
Similarity rule	A subset of products	OCL	Equality relationships	Results in reuse if selected

In addition to inferring values and reusing configuration decisions, using similarity rules, value changes will be automatically propagated into similar parts of the configuration. This allows keeping the configuration consistent after changing the value of a configurable parameter and without requiring extra effort. For example, as a result of selecting XtSemNum-Similarity, whenever the configuration engineer adds a new SEM to one of the Xmas trees (i.e., changes the number of SEMs in the Xmas tree) the inference engine automatically adds a new SEM to all other Xmas trees in the field.

9 Evaluation

To empirically evaluate our approach, we investigated two complete subsea products of our industry partner. These products, detailed in Table 2, are representative considering their size, types of components, and similarity specifications.

Table 2: An overview of the two investigated products.

*	**# XmasTrees**	**# SEMs**	**# Devices**	**# Configurable parameters ****
Product_1	9	18 (9 twin SEMs)	2360	29796
Product_2	14	28 (14 twin SEMs)	5072	56124
* The two products are very dissimilar with respect to their internal similarities and each represent one of two main types of subsea fields (scattered and clustered subsea fields). ** Total number of configurable parameters that need to be configured to create the software specification for the product.				

Similarity modeling. Generic software of the product family investigated in this case study contains 36 configuration units, which in total have 264 configurable features. To create a similarity feature model, we thoroughly studied both products and identified the similarities within each product. The resulting similarity feature model is a tree of depth four, with a total of 200 features, including 81 leaf features representing the similarity rules. These similarity rules have, in total, 423 equality relations that are defined in terms of classes and configurable features in the generic software model.

Similarity-based reuse. To create software products, we started by selecting the required similarity rules using the similarity feature model. The total number of selected similarity rules, and equality relations are reported, for each product, in Table 3. Among these similarity rules 12 are common between the two products, resulting in 110 equality relations in common. This relatively low number of common similarity rules reflects the fact that the chosen products are very dissimilar with respect to their internal similarities.

Table 3: Summary of similarity rules, and automated reuse in the two products.

	# Similarity rules	# Eq. Relation	# Auto. decisions	Reuse rate
Product_1	52	263	19289	0.647
Product_2	41	270	46801	0.834

To identify the effectiveness of our approach, we introduce a measure called *reuse rate*, which provides an insight into the percentage of the decisions that can be automatically inferred based on the applied similarity rules and the previously provided configuration decisions. The fourth column in Table 3 gives, for each product, the number of such decisions. Reuse rate, for each product, is calculated by dividing the number of automated decisions by the number of configurable parameters (last column in Table 2). As shown in the fifth column in Table 3, reuse rates for product_1 and product_2 are 0.647 and 0.834, respectively. It means that, for example in product_2, 83.4% of configuration decisions can be automatically made by the configuration tool using the similarity rules, and the user has to manually configure only 16.6% of the parameters. Given the very large number of configurable parameters, this result is of practical significance. In particular, assuming automated configuration decisions have similar complexity to manual ones, our results show that such an automation can save

more than 60% of the configuration effort in large-scale systems. Note that the 60% gain is calculated with respect to cases where no support for reuse is provided, not compared to the current situation at our industry partner where primitive support for reuse is provided through copy-and-paste mechanism.

Discussion. Modeling, in general, is manual and time consuming. This applies to our similarity modeling approach too. However, the effort that is put into creating similarity models is paid back because, (1) only one similarity model is created for each product family and is used during the configuration of all products, and (2) as our evaluation shows, a great portion of the configuration data can be automatically derived using similarity models, reducing the configuration effort. When the number of configurable parameters is very large–often in the thousands, as in many ICSs, the benefit of such similarity models can be substantial. This has shown to be clearly the case in our industrial case studies.

Hardware similarities that are the basis for automated reuse in our approach are present in many embedded software systems as well as distributed networked systems. Therefore, we expect our results to generalize to those domains, as well as to other ICSs with highly-symmetric hardware architectures.

10 Conclusion

This paper focuses on the automated similarity-based reuse of configuration data in families of integrated control systems (ICS). Individual ICS products, like many other embedded software systems, usually bear a high degree of similarity within their hardware structures, which results in internal similarities within their software configurations. In this paper, we propose an approach to model such internal similarities. As opposed to the commonalities in a product family that capture similarities among different products, internal similarities capture similarities among different parts of an individual product. In our similarity modeling approach, to enable automated reuse, we model internal similarities in terms of the elements in the generic model of the product family as a set of similarity rules using OCL. We use feature models to provide a user-level representation of similarity rules and the variabilities they

introduce. We evaluated the effectiveness of our approach using two product configurations from our industry partner. Our results show that an automated similarity-based approach to configuration reuse can save more than 60% of configuration decisions, and consequently, can reduce configuration effort. In future, we will conduct experiments with human subjects, to further evaluate the applicability of our approach.

Acknowledgments

This work was mostly supported by a grant from Det Norske Veritas (DNV). L. Briand was supported by a FNR PEARL grant. We are grateful to FMC Technologies Inc. for their help on performing the industrial case study.

Bibliography

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

[2] OCL: Object Constraint Language. http://www.omg.org/spec/OCL/2.2/, 2012.

[3] D. S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.

[4] R. Behjati, S. Nejati, T. Yue, A. Gotlieb, and L. Briand. Model-based automated and guided configuration of embedded software systems. In *ECMFA*, 2012.

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

[6] R. Behjati, T. Yue, L. Briand, and B. Selic. SimPL: a product-line modeling methodology for families of integrated control systems. *Submitted to Inf. Softw. Technol.*, 2011.

[7] J. Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *SPLC*, 2002.

[8] J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW*, 2008.

[9] K. Czarnecki. Mapping features to models: A template approach based on superimposed variants. In *GPCE*, 2005.

[10] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 2005.

[11] K. Czarnecki, S. Helsen, and U. W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 2005.

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

[13] F. Dordowsky and W. Hipp. Adopting software product line principles to manage software variants in a complex avionics system. In *SPLC*, 2009.

[14] A. Egyed. Instant consistency checking for the UML. In *ICSE*, 2006.

[15] C. Gillan, P. Kilpatrick, I. T. A. Spence, T. J. Brown, R. Bashroush, and R. Gawley. Challenges in the application of feature modelling in fixed line telecommunications. In *VaMoS*, 2007.

[16] H. Gomaa and M. E. Shin. Automated software product line engineering and product derivation. In *HICSS*, 2007.

[17] Ø. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, 2008.

[18] A. Hubaux, A. Classen, M. Mendonça, and P. Heymans. A preliminary review on the application of feature diagrams in practice. In *VaMoS*, 2010.

[19] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, 1990.

[20] 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.

[21] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J. M. Jézéquel. Weaving variability into domain metamodels. In *MODELS*, 2009.

[22] R. Rabiser, P. Grünbacher, and D. Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Inf. Softw. Technol.*, 52(3), March 2010.

[23] M. O. Reiser, R. Tavakoli Kolagari, and M. Weber. Unified feature modeling as a basis for managing complex system families. In *VaMoS*, 2007.

[24] 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.

[25] M. Steger, C. Tischer, B. Boss, A. Müller, O. Pertler, W. Stolz, and S. Ferber. Introducing PLA at bosch gasoline systems: Experiences and practices. In *Software Product Lines*, volume 3154, 2004.

[26] M. Svahnberg and J. Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, 11(6), November 1999.

[27] T. Ziadi and J. M. Jézéquel. Software product line engineering with the UML: Deriving products. 2006.