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

Razieh Behjati[1,2], Tao Yue[1], Lionel Briand[1,3]
[1]Certus Software V&V Center, Simula Research Laboratory, Norway
[2]University of Oslo, Oslo, Norway
[3]SnT Centre, University of Luxembourg, Luxembourg
{raziehb, tao, briand}@simula.no

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.
Keywords: Product configuration, Internal similarities, Model-based software engineering, UML/OCL, Feature Modeling.

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 [6]. 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 [6], 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.

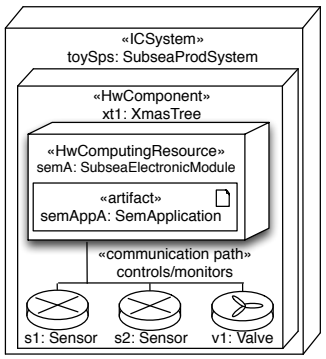

Fig. 1: Fragment of a simplified subsea production system.

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 [5, 4], 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 [19, 10] have been most commonly studied in the literature (e.g., [20, 16, 9]) 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., [24, 27, 17, 21]), 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

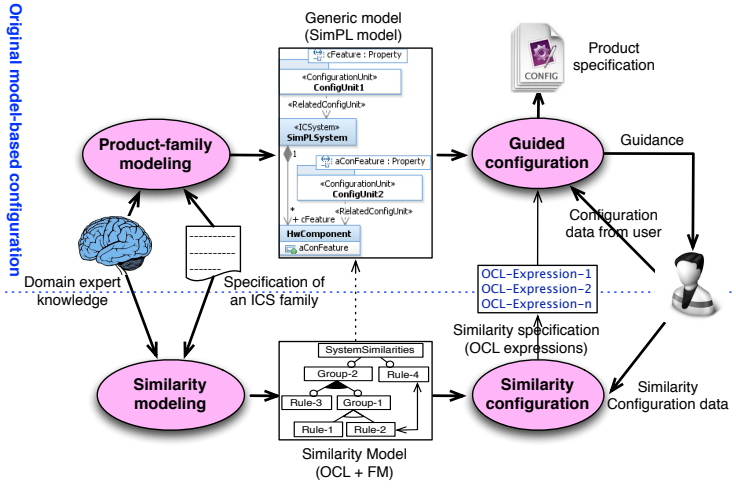

Fig. 2: An overview of our reuse-oriented configuration approach.

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.

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

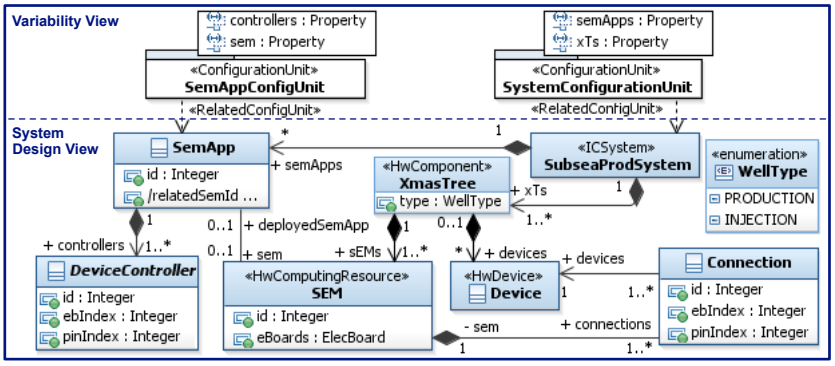

Fig. 3: A fragment of the SimPL model for the subsea production system.

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).

[1] This example is a sanitized fragment of a subsea production case study [6].

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[2] 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.

[2] 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.

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, 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[3]. 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.

```
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

[3] 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.

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. 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.

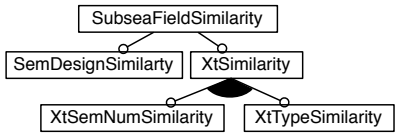

Fig. 4: A fragment of the similarity feature model for the subsea production systems family.

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 selected 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.

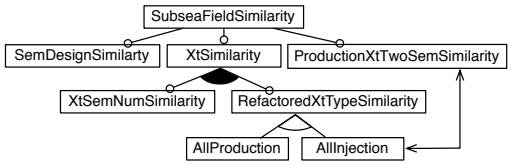

Fig. 5: Dependencies between similarity rules are modeled as dependencies between features.

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).

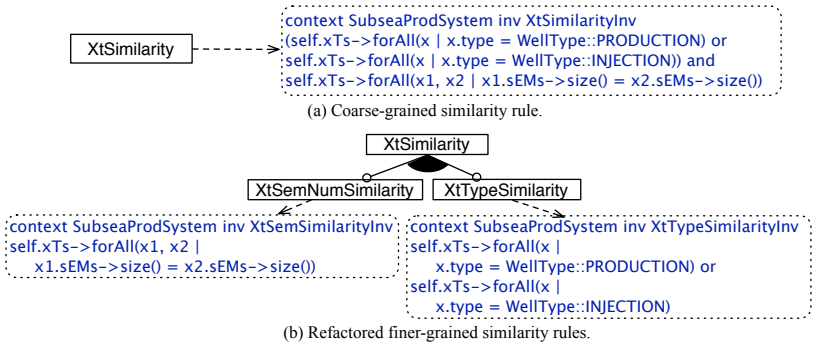

Fig. 6: Refactoring of a similarity rule.

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.

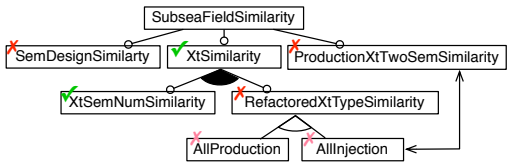

Fig. 7: Similarity feature model configured for a particular product.

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.

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 XtSemNumSimilarity, 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.

References

1. UML Superstructure Specification, v2.3, May 2010.
2. Object Constraint Language. http://www.omg.org/spec/OCL/2.2/, 2012.
3. Don S. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, pages 7–20, 2005.
4. Razieh Behjati, Shiva Nejati, Tao Yue, Arnaud Gotlieb, and Lionel Briand. Model-based automated and guided configuration of embedded software systems. In *ECMFA 2012, LNCS 7349*, pages 226–243, 2012.
5. Razieh Behjati, Tao Yue, Lionel Briand, and Bran Selic. SimPL: a product-line modeling methodology for families of integrated control systems. Submitted to Information and Software Technology Journal, 2011.
6. Razieh Behjati, Tao Yue, Lionel Briand, and Bran Selic. SimPL: a product-line modeling methodology for families of integrated control systems, Tech. Report 2011-14, Simula Research Lab. `http://simula.no/publications/Simula.simula.746`, 2011.
7. Jan Bosch. Maturity and evolution in software product lines: Approaches, artefacts and organization. In *Proceedings of the Second International Conference on Software Product Lines*, SPLC 2, pages 257–271, London, UK, UK, 2002. Springer-Verlag.
8. Jordi Cabot, Robert Clarisó, and Daniel Riera. Verification of uml/ocl class diagrams using constraint programming. pages 73–80, Washington, DC, USA, 2008. IEEE Computer Society.
9. Krzysztof Czarnecki. Mapping features to models: A template approach based on superimposed variants. In *GPCE 2005 - Generative Programming and Component Enginering. 4th International Conference*, pages 422–437. Springer, 2005.
10. Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. In *Software Process: Improvement and Practice*, 2005.
11. Krzysztof Czarnecki, Simon Helsen, and Ulrich W. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 2005.
12. Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product derivation in software product families: a case study. 74:173–194, January 2005.
13. Frank Dordowsky and Walter Hipp. Adopting software product line principles to manage software variants in a complex avionics system. In *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, pages 265–274, Pittsburgh, PA, USA, 2009. Carnegie Mellon University.
14. Alexander Egyed. Instant consistency checking for the uml. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 381–390, New York, NY, USA, 2006. ACM.
15. Charles Gillan, Peter Kilpatrick, Ivor T. A. Spence, T. John Brown, Rabih Bashroush, and Rachel Gawley. Challenges in the application of feature modelling in fixed line telecommunications. In *VaMoS*, pages 141–148, 2007.

16. Hassan Gomaa and Michael E. Shin. Automated software product line engineering and product derivation. In *Proceedings of the 40th Annual Hawaii International Conference on System Sciences*, HICSS '07, pages 285a–, Washington, DC, USA, 2007. IEEE Computer Society.
17. Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, Gøran K. Olsen, and Andreas Svendsen. Adding standardized variability to domain specific languages. In *SPLC*, pages 139–148, 2008.
18. Arnaud Hubaux, Andreas Classen, Marcílio Mendonça, and Patrick Heymans. A preliminary review on the application of feature diagrams in practice. In *VaMoS*, pages 53–59, 2010.
19. Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, 1990.
20. Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Gerard Jounghyun Kim, and Euiseob Shin. Form: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.
21. Brice Morin, Gilles Perrouin, Philippe Lahire, Olivier Barais, Gilles Vanwormhoudt, and Jean-Marc Jézéquel. Weaving variability into domain metamodels. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, pages 690–705. Springer-Verlag, 2009.
22. Rick Rabiser, Paul Grünbacher, and Deepak Dhungana. Requirements for product derivation support: Results from a systematic literature review and an expert survey. *Information & Software Technology*, 52(3):324–346, March 2010.
23. Mark-Oliver Reiser, Ramin Tavakoli Kolagari, and Matthias Weber. Unified feature modeling as a basis for managing complex system families. In *VaMoS*, volume 2007-01 of *Lero Technical Report*, pages 79–86, 2007.
24. André L. Santos, Kai Koskimies, and Antónia Lopes. A model-driven approach to variability management in product-line engineering. *Nordic J. of Computing*, 13:196–213, September 2006.
25. Mirjam Steger, Christian Tischer, Birgit Boss, Andreas Müller, Oliver Pertler, Wolfgang Stolz, and Stefan Ferber. Introducing pla at bosch gasoline systems: Experiences and practices. In *Software Product Lines*, volume 3154, pages 34–50. Springer, 2004.
26. Mikael Svahnberg and Jan Bosch. Evolution in software product lines: Two cases. *Journal of Software Maintenance*, 11(6):391–422, November 1999.
27. Tewfik Ziadi and Jean-Marc Jézéquel. Software product line engineering with the UML: Deriving products. pages 557–588, 2006.