Architecture-Level Configuration of Large-Scale Embedded Software Systems [⋆]

Razieh Behjati[1], Shiva Nejati[2], Lionel Briand[2]
[1]Certus Software V&V Center, Simula Research Laboratory, Norway
[2]SnT Centre, University of Luxembourg, Luxembourg
`raziehb@simula.no, {shiva.nejati, lionel.briand}@uni.lu`

Abstract. Configuration in the domain of integrated control systems (ICS) is largely manual, laborious, and error-prone. In this paper, we propose a model-based configuration approach that provides automation support for reducing configuration effort and the likelihood of configuration errors in the ICS domain. We ground our approach on component-based specifications of ICS families. We then develop a configuration algorithm using constraint satisfaction techniques over finite domains to generate products that are consistent with respect to their ICS family specifications. We reason about the termination and consistency of our configuration algorithm analytically. We evaluate the effectiveness of our configuration approach by applying it to a real subsea oil production system. Specifically, we have rebuilt a number of existing verified product configurations of our industry partner. Our experience shows that our approach can automatically infer up to 50% of the configuration decisions, and reduces the complexity of making configuration decisions.
Keywords: Model-based product-line engineering, Product configuration, Consistent configuration, Constraint satisfaction techniques, Formal specification, UML/OCL.

1 Introduction

Integrated control systems (ICS) are large-scale heterogeneous systems that are used in many industry sectors where hardware and software systems are integrated to control production processes and to ensure safety aspects. Examples include the energy, automotive, and avionics industries. In most situations, product-line engineering approaches [46, 55, 39] are applied to develop integrated control systems. Software development, in this context, is done through configuring a *reference architecture*, which provides a common, high-level, and customizable structure for all members of the product family [46].

In our earlier work [7, 8], we studied the challenges of architecture-level software configuration in families of integrated control systems. Findings reported in [7] and in other published studies [24, 44] show that, in general, software configuration in large-scale embedded software systems is a laborious and error-prone

[⋆] This paper is an extension of a conference paper "Model-Based Automated and Guided Configuration of Embedded Software Systems" published in proceedings of the Eighth European Conference on Modelling Foundations and Applications (ECMFA), 2012 [6].

task. This is due, in large part, to the complexity of such systems and inadequacies in the adoption of product-line engineering approaches. The latter is a result of the lack of concise abstractions for product families and insufficient automation support for the configuration process – both crucial for a product-line engineering approach to succeed [37].

Software configuration has been previously studied in the area of software product lines, where support for configuration largely concentrates on resolving variabilities in feature models [35, 36] and their extensions [18, 20]. Feature models, however, are not easily amenable to capturing all kinds of architectural variabilities in the ICS domain. Furthermore, existing configuration approaches either do not enable instant validation of configurations (e.g., [34]), or their notion of configuration and their underlying mechanism are different from ours, and hence, not directly applicable to our problem domain (e.g., [41]).

To overcome the configuration challenges in the context of integrated control systems, we propose a model-based and semi-automated approach for the architecture-level configuration of software in such systems. Figure 1 depicts an overview of our configuration approach. As shown in this figure, our approach has two major steps: the product-family modeling step and the semi-automated configuration step. In the first step (the product-family modeling step), a model of the reference architecture of the product family is created. This model is then used as an input to the second step (the semi-automated configuration step) to provide automation support for creating reliable product configurations.

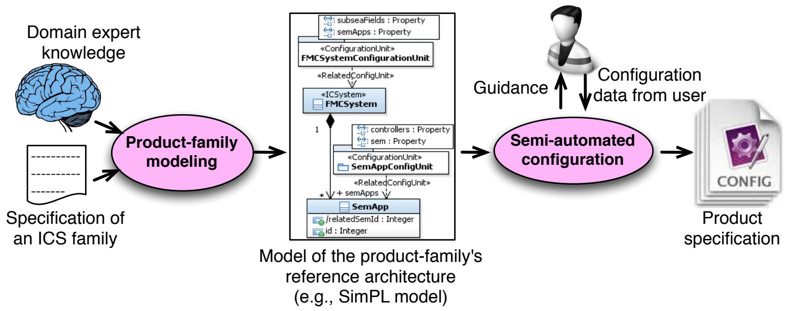


Fig. 1. An overview of our model-based semi-automated configuration approach.

In this paper, we focus on the second step and propose a configuration framework that enables us to reduce the configuration effort, the complexity of decision making, and the likelihood of human errors, while ensuring that the configured products are consistent with the input reference architectures. We utilize constraint satisfaction techniques over finite domains [29, 16] to provide the automation support for our configuration framework. The core idea of our approach has been presented in [6]. The present article extends and refines our earlier work, making the following contributions:

1. We formalize the notion of component-based reference architectures for ICS families. Specifically, our formalization characterizes reference architectures,

products, the definition of *consistency* of an individual reference architecture and its related products, and various types of variabilities that arise in ICS families (i.e., attribute, cardinality, type, topology). We further provide a (linear) mapping, casting configuration of ICS families to a constraint satisfaction problem over finite domains.

2. We propose a configuration algorithm for the semi-automated configuration step in Figure 1. Our algorithm supports the configuration of all types of variabilities (i.e., attribute, cardinality, type, topology). We reason about the consistency and termination of our algorithm. Specifically, our proof of consistency shows that our algorithm, when given a consistent reference architecture model, can always generate a complete and consistent product. Our proof of termination shows that the algorithm always terminates in finite time provided that every variability in the input reference architecture can be resolved using a finite set of variants.
3. We provide an implementation of our configuration algorithm based on constraint satisfaction techniques. Our tool uses the SICStus Prolog constraint solver [15, 4]. The tool interactively guides engineers to make configuration decisions and automates some of the decisions.
4. We have applied our prototype tool to configure a family of subsea oil production systems. We rebuilt three configurations from this product family. The rebuilt configurations contained 343, 2830, and 5397 configurable parameters, and were configured in a total of 4735 configuration iterations. Our experiments show that, for the subjects in our experiment, our approach can automate up to 50% of configuration decisions, and within nine seconds provide the user with accurate valid domains, which on average shrink by 38% with each configuration iteration, thus simplifying configuration decisions.

For the first step of Figure 1, in an earlier work [7], we proposed a modeling methodology, named SimPL, characterizing the notion of reference architecture in the ICS domain. This methodology describes an ICS family in terms of constructs from the Unified Modeling Language (UML 2) [1], and its extension for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [2]. In the current implementation of our configuration framework, input reference architectures are SimPL models. Furthermore, the running examples provided throughout the paper are excerpts of SimPL models.

Section 2 introduces the industrial background of our research and precisely formulates the problem we aim to address in this paper. This section, further, describes the current product configuration practice in the integrated control systems domain under study. In Section 3, we describe the notion of reference architecture in the ICS domain and briefly introduce the SimPL methodology and its main concepts. We introduce our running example in this section. A formal specification of the notion of reference architecture in the ICS domain together with formal definitions of consistency is presented in Section 4. We describe the main configuration algorithm in Section 5. Consistency related aspects of the configuration process are presented and formalized in Section 6. Characteristics of our semi-automated configuration approach are described and proved in Section 7. Our prototype configuration tool and results of our evaluation of the semi-automated configuration approach are presented in Sections 8 and 9. In Section 10, we discuss the applicability and generalizability of both our modeling and our configuration solutions. Section 11 discusses the related work and Section 12 concludes the paper.

2 Configuration of ICSs: Practice and Problem Definition

Figure 2 shows a simplified model of a fragment of a subsea oil production system produced by one of our industry partners, used in this paper as a case study. 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 (e.g., SemApplications) for controlling and monitoring devices in a subsea well.

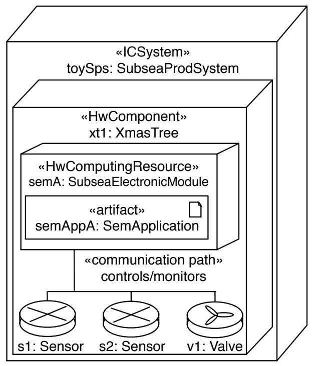


Fig. 2. A fragment of a simplified oil production system.

While Figure 2 shows a few types of generic or configurable components only (e.g., XmasTree, SemApplication), real-world ICSs typically consist of hundreds of configurable components. In the ICS domain, customized systems, which are created through configuration, consist of thousands of components and tens of thousands of interdependent parameters that need to be configured individually. For example, the product family that we studied at our industry partner contains more than one hundred configurable software and hardware components. Two representative products derived from this product family consisted of 2360 and 5072 hardware devices, and were generated by configuring 29796 and 56124 parameters, respectively.

Product configuration is an essential activity in ICS development. It involves configuration of both software and hardware components. 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 may need to be further decomposed and configured. Configuration stops once the type and the number of all components and the values of their configurable parameters are determined.

For example, software configuration for a family of subsea oil 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 in charge of configuration) is typically provided with a hardware configuration plan. However, he has to manually check if the resulting software configuration conforms to the given hardware plan, and that it complies with 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 consistency checking, human errors such as incorrectly entered values are usually discovered very late in the development life-cycle, making localizing and fixing such errors unnecessarily costly.

In short, in the context of ICS development, existing configuration support frequently faces the following shortcomings [8]: (1) There is no automated support to help engineers configure new products, and further, reusing configuration data from old products is done in an old-fashioned way (e.g., by copy-and-paste). (2) Instant configuration checking and verification of partially-specified configurations are not supported. (3) Engineers are not provided with sufficient interactive guidance throughout the configuration process. In this paper, we propose a model-based and semi-automated configuration solution to address the aforementioned configuration challenges. Our solution interactively and instantly validates configuration decisions and identifies implications of each decision. To do so, it uses a model of the reference architecture of the product family.

3 The SimPL methodology

A major asset of a product family is its *reference architecture*, which provides a common and high-level structure for all members of the product family [46]. A reference architecture specifies different types of reusable components that may exist in some members of the product family. Each *component type* may have relationships with other component types, and has a number of *configurable features* through which it defines a number of variability points.

As shown in Figure 1, the reference architecture of a product family is one of the inputs to the configuration process. During configuration and using the user-provided configuration data, instances of component types are created and configured. An instance of a component type is called a *component*, and consists of a number of *configurable parameters*. Configurable parameters are the variables that collect all the necessary configuration information throughout the configuration process. Each configurable parameter is an instance of a configurable feature. A component is configured by assigning values to its configurable parameters.

In our earlier work [7, 8], we proposed the SimPL methodology to create models of reference architectures for product families in the ICS domain. The running examples in this paper are presented using the SimPL methodology. The SimPL methodology organizes a model of a reference architecture into two main views: the *system design view*, and the *variability view*. Figure 3 shows the SimPL model of the reference architecture for a family of ICSs. The system design view and the variability view are shown in the lower and the upper parts of the figure, respectively.

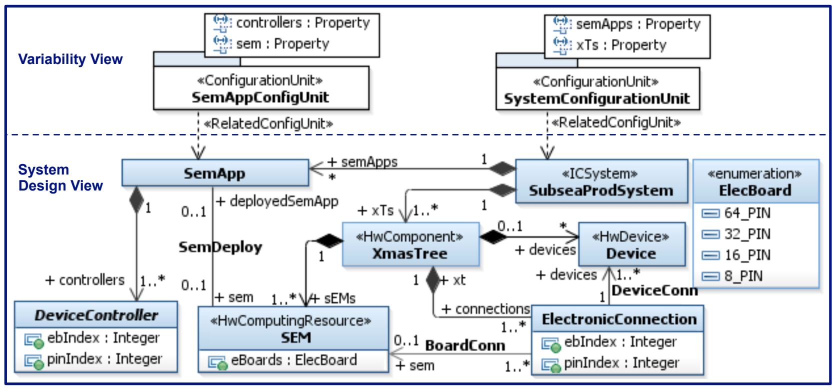


Fig. 3. A fragment of the SimPL model (reference architecture) for the subsea oil production family.

The system design view presents both hardware and software entities of the system and their relationships using the UML class diagram notation [1]. A class in the system design view represents a component type. Relationships between classes specify how components can be connected and structured. Each SimPL model contains a special class stereotyped by «ICSystem» representing the topmost component type (e.g., SubseaProdSystem in Figure 3). Each product derived from a SimPL model has one instance of the topmost component type.

The variability view captures the set of system variabilities using a collection of *template packages*. Each template package, in this context, is named a *configuration unit* and is related to exactly one class in the system design view. A template parameter of a template package represents a configurable feature and describes a variability in the value, type, or cardinality of a property defined in the context of the corresponding class.

In addition to the two views described above, each SimPL model has a repository of OCL expressions [3]. These OCL expressions specify constraints among the values, types, or cardinalities of different properties of different classes. These OCL constraints are part of the product family commonalities and must hold for all the products in the family. Table 1 summarizes the product-line modeling concepts and their equivalent constructs in the SimPL methodology.

In the rest of this section, we first present a fragment of a subsea product-family model, which is used as our running example in the rest of the paper. Then, we present a model of a small subsea product derived from that product-family. Finally, we present a classification of configurable features. Based on this classification we then describe the configuration process in subsequent sections.

Table 1. Product-line modeling (PLM) concepts and their equivalent constructs in SimPL.

PLM Concept	SimPL Construct
Reference architecture	SimPL model
Component types	Classes & configuration units
Configurable features	Properties & template parameters
Components	Objects
Configurable parameters	Object parameters
Constraints	OCL expressions

3.1 Reference architecture for a family of subsea systems

Figure 3 shows a fragment of a SimPL model representing the simplified reference architecture of a family of subsea oil production systems[1] described in Section 2. In a subsea oil production system, i.e., SubseaProdSystem, 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.

The variability view in the SimPL methodology is a collection of template packages. The upper part in Figure 3 shows a fragment of the variability view for the subsea oil production family. In order to remain concise, we have shown only two template packages in the figure, which should be enough for the reader to understand the underlying principles. The package SystemConfigurationUnit represents the configuration unit related to the class SubseaProdSystem. Template parameters of this package specify the configurable features of the component type modeled by the class SubseaProdSystem. These configurable features are: the number of XmasTrees (xTs), and the number of SEM applications (semApps).

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

context ElectronicConnection inv BoardIndRange
ebIndex >= 0 and ebIndex < 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

[1] This example is a sanitized fragment of a subsea oil production case study [8].

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 electronic boards on its SEM.

3.2 A subsea oil production system

Figure 4 shows a model of a small subsea oil production system created by configuring the reference architecture given in Figure 3. The product shown in Figure 4 is created by configuring a total of ten configurable parameters. The topmost component in this product is an instance of the class SubseaProdSystem and is named toySps. This component is configured by setting both the number of its Xmas trees (i.e., XmasTree) and the number of its SEM applications (i.e., SemApp) to one. The components xt1 and semAppA are created as the result of configuring toySps. The Xmas tree xt1 is configured to contain one subsea electronic module (i.e., semA) and one device (i.e., s1). The subsea electronic module semA has two electronic boards. The device s1 has a timeout of 100 ms and is connected to the first pin of the first electronic board of semA as shown in semAs1 (i.e., ebIndex = 0, and pinIndex = 0). Finally, the SEM application semAppA is deployed to the subsea electronic module semA.

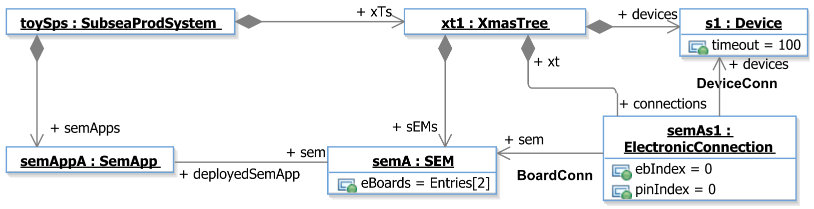


Fig. 4. Model of a simple product belonging to the product-family model in Fig. 3.

Configuration of a real subsea oil production system is similar to the configuration of the product described above, but involves assigning values to tens of thousands of configurable parameters.

3.3 Classification of configurable features

As mentioned earlier, a configurable feature of a component type describes a variability in the value, type, or cardinality of a property of that component type. We classify configurable features into four groups according to the type of the variability they express [8]. Figure 5 shows this classification. Each type of configurable features in this classification is configured differently and its configuration carries different consequences. For example, configuring the cardinality of a property may result in creating new components in the product (e.g., creating a new XmasTree instance), while configuring the value of a property does not have such a consequence. In this section, we use the SimPL methodology and its constructs to briefly describe each type of configurable features. Details on the configuration of each type of configurable features are presented later in the paper.

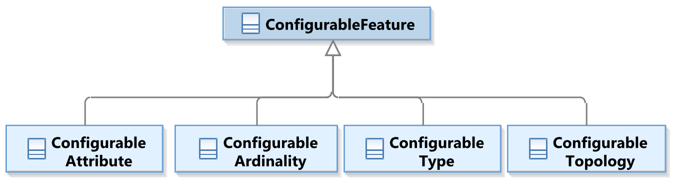


Fig. 5. Classification of configurable features.

1. **Configurable attribute.** A configurable attribute is represented by a configurable feature specifying a variability in the value of an attribute of a class in a SimPL model. For example in Figure 3, the value of attribute pinIndex might be different for each instance of class ElectronicConnection, and therefore this attribute introduces a variability that is represented by a configurable feature of type attribute (i.e., a configurable attribute).
2. **Configurable cardinality.** A configurable cardinality is represented by a configurable feature specifying a variability in the cardinality of a set of objects, attributes, or object pointers (i.e., denoting association ends in UML). For example in Figure 3, the configurable feature modeled by the template parameter xTs in SystemConfigurationUnit is a configurable cardinality.
3. **Configurable type.** A configurable type is represented by a configurable feature specifying a variability in the concrete type of an object. In a reference architecture, configurable types are tightly coupled with generalization hierarchies. Figure 6 shows a generalization hierarchy consisting of the abstract class DeviceController and its subclasses. We use the term *generic* class (or generic component type) to refer to any class (or component type) that has a number of subclasses (or subtypes). Each instance of DeviceController contained by an instance of SemApp must be typed by a concrete subtype of DeviceController (i.e., either ValveController, SensorController, or Choke-Controller). A configurable feature (e.g., controllers) is therefore needed in SemApp to specify the variability in the concrete type of its contained DeviceControllers. We refer to such a configurable feature as a configurable type.
4. **Configurable topology.** Each component in a product may have connections to other components. Connections between components of a product form the topology of that product. Such a topology can vary from one product to another. To achieve a different topology, connections between components are configured differently. A connection is configured by specifying the components connected to each of its ends. A connection between two components is an instance of an association in the SimPL model. Therefore, association ends in the SimPL model introduce variability in topology. For an association end that introduces a variability in topology, a configurable feature is defined in the SimPL model. We use the term configurable

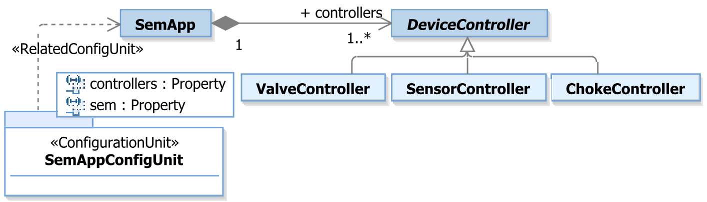


Fig. 6. A generalization hierarchy that renders a configurable type.

topology to refer to such a configurable feature. Figure 7 shows a variability in topology. Note that both variants in this figure have the same set of components.

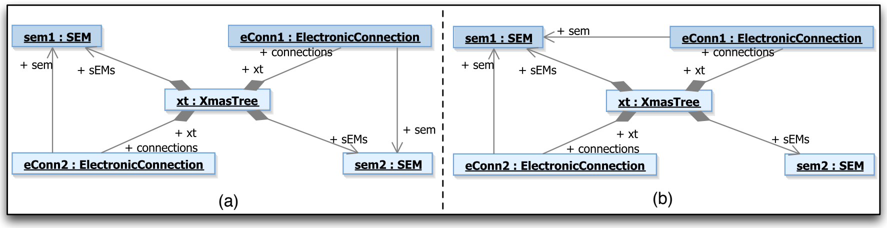


Fig. 7. A variability in topology. Variants (a) and (b) have identical components, but different connections between these components.

4 Formal specifications

In this section, we provide formal specifications for reference architectures that are modeled using the SimPL methodology. Further, we provide formal specifications for the products derived from such reference architectures. Then, we precisely define the notion of product consistency, which is a central concept in our approach to configuration. The definitions provided in this section, are used in the following sections (Sections 5-7) to define the configuration process and the functionalities required for checking and ensuring the consistency of products with respect to their reference architectures, and to prove the main characteristics of our semi-automated configuration approach.

4.1 Reference architecture

In the SimPL methodology, the reference architecture is a class model that defines different types of reusable components, their relationships, and the configurable features introduced by the components and their relationships. We formally define such a reference architecture as follows:

Definition 1 (Reference architecture). *A reference architecture RA represents a family of systems, and is defined as a sextuple $(CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$.*

- *CT is a set of component type names,*
- *$\prec$ is a decomposition relation over CT: $\prec \subseteq CT \times CT \times \mathbb{N}_0 \times \mathbb{N}_*$*[2]*,*
- *$\mathcal{G}$ is a partial function over CT ($\mathcal{G} : CT \nrightarrow CT$), defining a generalization relation,*
- *N is a set of association names,*
- *$\mathcal{A}$ is a function over N, defined as $\mathcal{A} : N \to CT \times \mathbb{N}_0 \times \mathbb{N}_* \times CT \times \mathbb{N}_0 \times \mathbb{N}_*$ to represent association relations, and*
- *Φ is a set of constraints that should hold for all members of the product family.*

Each member (ct, ct', l, u) in the decomposition relation $\prec$ specifies that each instance of the component type named ct consists of at least l and at most u instances of the component type named ct'. We use $l..u$ to denote the multiplicity of this decomposition. We refer to ct as the *source* of the decomposition and to ct' as the *target* of the decomposition. Each member (ct', ct) in $\mathcal{G}$ specifies that the component type named ct' is a subtype of ct. Note that the definition of $\mathcal{G}$ does not allow multiple inheritance. We define a function $subtypes : CT \to \mathcal{P}(CT)$ to map each component type to the set of all of its subtypes. Specifically, let ct be a component type in CT, then $subtypes(ct)$ is the set of all component types ct' such that $(ct', ct) \in \mathcal{G}$. For each association name $n \in N$, $\mathcal{A}(n)$ is a sextuple (ct, l, u, ct', l', u') specifying that each instance of ct is linked, with a link named n, to $l'..u'$ instances of ct'. Similarly, each instance of ct' is linked, with a link named n, to $l..u$ instances of ct. The sets CT and N, the relations $\prec$ and $\mathcal{G}$, and the association function $\mathcal{A}$ are created from the classes and the relationships between classes in the SimPL model of the product family.

Example 1. The SimPL model in Figure 3 can be represented by $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$. Figure 8 shows how each element of RA is defined for the SimPL model in Figure 3.

As shown in Figure 8, the set CT consists of the names of the UML classes in the SimPL model. Each element of the decomposition relation $\prec$ maps to a composition association in the SimPL model. The set $\mathcal{G}$ is empty, as there are no generalization hierarchies in the SimPL model in Figure 3. The set N contains the association names in the SimPL model, and the function $\mathcal{A}$ defines the association relations for the association names in N. For example, $\mathcal{A}(\mathsf{DeviceConn})$ is the sextuple $(\mathsf{ElectronicConnection}, 1, 1, \mathsf{Device}, 1, *)$. We use the symbol $*$ to denote there is no fixed upper bound for an integer interval (i.e., $*$ can be replaced by any $k \in \mathbb{N}_1$). Finally, the set Φ contains a set of boolean formulas, including two formulas representing OCL expressions BoardIndRange and PinRange defined in Section 3.

[2] Let $\mathbb{N}_1$ denote the set of positive integer numbers. Then, we define $\mathbb{N}_0 = \mathbb{N}_1 \cup \{0\}$, and $\mathbb{N}_* = \mathbb{N}_0 \cup \{*\}$, where $*$ is a symbol that can be replaced by any $k \in \mathbb{N}_0$.

CT	$=$ {SubseaProdSystem, SemApp, Device, XmasTree, SEM, DeviceController, ElectronicConnection}
$\prec$	$=$ {(SubseaProdSystem, SemApp, 0, *), (XmasTree, Device, 0, *), (SubseaProdSystem, XmasTree, 1, *), (XmasTree, SEM, 1, *), (XmasTree, ElectronicConnection, 1, *), (SemApp, DeviceController, 1, *)}
$\mathcal{G}$	$= \emptyset$
N	$=$ {DeviceConn, BoardConn, SemDeploy}
$\mathcal{A}(DeviceConn)$	$=$ (ElectronicConnection, 1, 1, Device, 1, *)
$\mathcal{A}(BoardConn)$	$=$ (ElectronicConnection, 1, *, SEM, 0, 1)
$\mathcal{A}(SemDeploy)$	$=$ (SemApp, 0, 1, SEM, 0, 1)
Φ	$= \{\phi_{\mathsf{BoardIndRange}}, \phi_{\mathsf{PinRange}}\}$

Fig. 8. Mathematical structure for the SimPL model in Figure 3.

In Definition 1, each member of the set CT denotes a component type. A component type in the reference architecture represents a reusable component and is formally defined as follows:

Definition 2 (Component type). *Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture. Each component type $ct \in CT$ has a set of configurable features. We use F_{ct} to denote the set of configurable features of ct.*

Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture and let $ct \in CT$ be a component type in RA. We partition the set F_{ct} of configurable features of ct into six disjoint sets $AF_{ct}, AC_{ct}, DC_{ct}, SC_{ct}, DT_{ct}$ and TF_{ct}, denoting respectively the sets of configurable attributes, configurable attribute cardinalities, configurable decomposition cardinalities, configurable association cardinalities, configurable types, and configurable topologies (See Section 3.3 for the definition of different types of configurable features). Each member of the set AF_{ct} of configurable attributes is typed by either integer, boolean, or a user-defined enumeration (i.e., a finite set of tags). Members of the sets AC_{ct}, DC_{ct} and SC_{ct} are all configurable features of type cardinality and take their values from integer intervals (i.e., $(l, u) \in \mathbb{N}_0 \times \mathbb{N}_*$). We use the term *cardinality feature* to refer to any member of AC_{ct}, DC_{ct} or SC_{ct}. Each configurable feature in DT_{ct} represents a configurable type and takes its value from a subset of CT. Finally, each configurable feature in TF_{ct} represents a variability in topology and takes its value from a subset of instances of the component types defined in CT.

We define a function $type : F_{ct} \to UT$ to map each configurable feature in F_{ct} to its type. In the definition of the function *type*, the set UT denotes a universal set of types (i.e., $UT = \{\mathsf{bool}, \mathsf{int}\} \cup E \cup (\mathbb{N}_0 \times \mathbb{N}_*) \cup \mathcal{P}(CT) \cup \mathcal{P}([\![CT]\!])$, where E denotes the set of all user-defined enumerations in the SimPL model of the reference architecture RA, and $[\![CT]\!]$ denotes the set of all instances of all the component types in CT). Table 2 summarizes the six sets of configurable features described above.

Table 2. Different types of configurable features.

Name	**Set notation**	**Type**
Configurable attribute	AF_{ct}	$\{\mathsf{bool}, \mathsf{int}\} \cup E$
Configurable attribute cardinality	AC_{ct}	$Range(\mathbb{N}_0)$
Configurable decomposition cardinality	DC_{ct}	$Range(\mathbb{N}_0)$
Configurable association cardinality	SC_{ct}	$Range(\mathbb{N}_0)$
Configurable type	DT_{ct}	$\mathcal{P}(CT)$
Configurable topology	TF_{ct}	$\mathcal{P}([\![CT]\!])$
$Range(\mathbb{N}_0) = \{(l, u) \| l \in \mathbb{N}_0, u \in \mathbb{N}_0, l \leq u\}$		

Note that F_{ct} contains both the configurable features that are directly defined in *ct*, and the ones that are inherited from supertypes of *ct*. The same rule applies to the sets $AF_{ct}, AC_{ct}, DC_{ct}, SC_{ct}$, DT_{ct} and TF_{ct}. In our approach, we do not allow multiple inheritance. This implies that each component type has at most one direct supertype, but it may as well have several indirect supertypes.

In the following, we define several functions to specify the dependencies between each type of configurable features and the elements of the reference architecture RA.

– Each member of the set AC_{ct} of configurable attribute cardinalities denotes the variability in the cardinality of an attribute of *ct*. Recall that AF_{ct} is a set of configurable attributes. Each feature in AC_{ct} is related to a member of the set AF_{ct}. We define a function $atr : AC_{ct} \to AF_{ct}$ to map each feature in AC_{ct} to its related member of AF_{ct}.
– Each member of the set DC_{ct} of configurable decomposition cardinalities denotes the variability in the cardinality of a decomposition of *ct*. We define a function $chl : DC_{ct} \to CT$ to map each feature $f \in DC_{ct}$ to the target of its corresponding decomposition. Specifically, let f be a feature in DC_{ct}, then $(ct, chl(f), l, u) \in \prec$ is the corresponding decomposition, and we have $l \neq u$.
– Each member of the set SC_{ct} of configurable association cardinalities denotes the variability in the cardinality of an association in which *ct* is an endpoint. We define a function $asc : SC_{ct} \to N$ to map each cardinality feature f in SC_{ct} to the corresponding association name n in N. Specifically, let f be a cardinality feature in SC_{ct}, then $asc(f)$ identifies its corresponding association name, and we have $\mathcal{A}(asc(f)) = (ct, l, u, ct', l', u')$ or $\mathcal{A}(asc(f)) = (ct', l', u', ct, l, u)$ and in both cases $l' \neq u'$.
– Each member of the set DT_{ct} of configurable types denotes a type variability introduced by a decomposition relationship between *ct* and a generic component type (i.e., a component type with a number of subtypes). We define a function $chlT : DT_{ct} \to CT$ to map each feature $f \in DT_{ct}$ to the target of the corresponding decomposition. Specifically, let f be a feature in DT_{ct}, then $(ct, chlT(f), l, u) \in \prec$ is the corresponding decomposition, and we have $subtypes(chl(f)) \neq \emptyset$.

- Each member of the set TF_{ct} of configurable topologies denotes a variability in the product topology. We define a function $ascT : TF_{ct} \to N$ to map each configurable topology to the name of the corresponding association. Specifically, let f be a feature in TF_{ct}, then $ascT(f) \in N$ identifies the corresponding association name, and we have $\mathcal{A}(ascT(f)) = (ct, l, u, ct', l', u')$ or $\mathcal{A}(ascT(f)) = (ct', l', u', ct, l, u)$.

Example 2. Each UML class in the SimPL model in Figure 3 represents a component type. Configurable features of each component type are defined based on its attributes, its relations to other classes, and the template parameters of its related configuration unit. Figure 9 shows mathematical representations for two component types (namely, SubseaProdSystem and SemApp) of the SimPL model in Figure 3. As shown in Figure 9, the component type SubseaProdSystem declares two configurable features, xTs_{dc} and $\mathsf{semApps}_{dc}$, both of type decomposition cardinality. These configurable features correspond to the template parameters of the configuration unit SystemConfigurationUnit associated with the class SubseaProdSystem (Figure 3). The feature xTs_{dc} represents the variability in the cardinality of the decomposition relationship between SubseaProdSystem and XmasTree. Therefore, we have $chl(\mathsf{xTs}_{dc}) = \mathsf{XmasTree}$. Similarly, for $\mathsf{semApps}_{dc}$ we have $chl(\mathsf{semApps}_{dc}) = \mathsf{SemApp}$. Configurable features of the component type SemApp are also defined in Figure 9 in the same manner.

$$
\begin{array}{ll}
\text{SubseaProdSystem (SPS)} & \\
\quad F_{\mathsf{SPS}} & = \{\mathsf{xTs}_{dc}, \mathsf{semApps}_{dc}\} = DC_{\mathsf{SPS}} \\
\quad AF_{\mathsf{SPS}} & = AC_{\mathsf{SPS}} = SC_{\mathsf{SPS}} = DT_{\mathsf{SPS}} = TF_{\mathsf{SPS}} = \emptyset \\
\quad chl(\mathsf{xTs}_{dc}) & = \mathsf{XmasTree} \\
\quad chl(\mathsf{semApps}_{dc}) & = \mathsf{SemApp} \\
\text{SemApp (SA)} & \\
\quad F_{\mathsf{SA}} & = \{\mathsf{controllers}_{dc}, \mathsf{controllers}_{dt}, \mathsf{sem}_{tf}\} \\
\quad DC_{\mathsf{SA}} & = \{\mathsf{controllers}_{dc}\} \\
\quad DT_{\mathsf{SA}} & = \{\mathsf{controllers}_{dt}\} \\
\quad TF_{\mathsf{SA}} & = \{\mathsf{sem}_{tf}\} \\
\quad AF_{\mathsf{SA}} & = AC_{\mathsf{SA}} = SC_{\mathsf{SA}} = \emptyset \\
\quad chl(\mathsf{controllers}_{dc}) & = \mathsf{DeviceController} \\
\quad chlT(\mathsf{controllers}_{dt}) & = \mathsf{DeviceController} \\
\quad acsT(\mathsf{sem}_{tf}) & = \mathsf{SemDeploy}
\end{array}
$$

Fig. 9. Mathematical structure for component in the SimPL model in Figure 3.

Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture. We use F_{CT} to denote the set of all configurable features defined by RA. Specifically, $F_{CT} = \bigcup_{ct \in CT} F_{ct}$.

4.2 Product

In our approach, a product represents an integrated control system and specifies it through its components, the connections between the components, the configurable parameters of the components, and the values of those configurable parameters. The formal specification of a product is given in Definition 3.

Definition 3 (Product). *A product P is denoted by a quadruple $(C, \preceq, N, L)$, where:*

- *C is a set of component names,*
- *$\preceq$ is a decomposition relation over C : $\preceq \subseteq C \times C$,*
- *N is a set of link names, and*
- *$L \subseteq C \times N \times C$ denotes a set of named links between the components in C.*

A decomposition $(c, c') \in \preceq$ specifies that component c encompasses component c' (i.e., c' is a subcomponent of c). We define a function $parts : C \rightarrow \mathcal{P}(C)$ to map each component to the set of its subcomponents. Specifically, let c be a component in C, then $parts(c) \subset C$ is the set of all components that are in a decomposition relationship with c (i.e., $\forall c' \in parts(c).(c, c') \in \preceq$). A named link $(c, n, c') \in L$ specifies that a link, named n, connects the component named c to the component named c'.

Example 3. The object diagram in Figure 4 specifies a simple product. According to Definition 3, we denote this product using the quadruple $P = (C, \preceq, N, L)$. Figure 10 depicts the mathematical representation of the product given in Figure 4. As shown in Figure 4, xt1 has three subcomponents: s1, semAs1, and semA. Therefore, $parts(\text{xt1}) = \{\text{s1, semAs1, semA}\}$.

$$
\begin{aligned}
C &= \{\text{toySps, semAppA, xt1, semA, s1, semAs1}\} \\
\preceq &= \{(\text{toySps, semAppA}), (\text{toySps, xt1}), (\text{xt1, s1}), \\
&\quad (\text{xt1, semA}), (\text{xt1, semAs1})\} \\
N &= \{\text{DeviceConn, BoardConn, SemDeploy}\} \\
L &= \{(\text{semAs1, DeviceConn, s1}), (\text{semAs1, BoardConn, semA}), \\
&\quad (\text{semAppA, SemDeploy, semA})\}
\end{aligned}
$$

Fig. 10. Mathematical representation for the product in Figure 4.

A product specifies an individual member of a product family and is derived from the reference architecture describing the product family. Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture. We use $[\![RA]\!]$ to denote the set of all products that can be derived from the product family represented by RA. Let $P = (C, \preceq, N, L)$ be a product derived from RA (i.e., $P \in [\![RA]\!]$). Each component c in C is an instance of a concrete component type ct in CT. We use $[\![ct]\!]$ to denote the set of all instances of the component type ct. We define a function $\mathcal{T} : C \rightarrow CT$ to map each component c to its corresponding concrete component type. The concrete type of a component can have a number of super types. Let ct be a component type and ct' be a super type of ct (i.e., $(ct, ct') \in \mathcal{G}$), then $[\![ct]\!] \subseteq [\![ct']\!]$. We define the function $types : C \rightarrow \mathcal{P}(CT)$ to map each component to the set of all of its component types. Specifically, let c be a component in C, then $types(c) \subset CT$ contains all component types ct such that $c \in [\![ct]\!]$. In other words, for a component $c \in C$, members of $types(c)$ are the concrete type of c (i.e., $\mathcal{T}(c)$) and all the direct and indirect super types of $\mathcal{T}(c)$. The set N of link names of P is the same as the set of association names of RA, and each link between two components of P is an instance of an association relation defined by $\mathcal{A}$ in RA. A precise specification of the conformance of a product to its reference architecture is discussed in Section 4.3.

Example 4. The product given in Figure 4 is derived from the SimPL model in Figure 3. The concrete type of the component named $\mathsf{xt1}$ is $\mathsf{XmasTree}$. Therefore, we have $\mathcal{T}(\mathsf{xt1}) = \mathsf{XmasTree}$. Since $\mathsf{XmasTree}$ has no super types according to Figure 4, $types(\mathsf{xt1})$ has only one member which is $\mathsf{XmasTree}$.

Definition 4 (Component). *Let $P = (C, \preceq, N, L)$ be a product. Each component c in C has a set of configurable parameters. We use P_c to denote the set of configurable parameters of component c. Furthermore, each configurable parameter in the set P_c is an instance of a configurable feature in the set $F_{\mathcal{T}(c)}$.*

Example 5. As shown in Figure 4, the component toySps is an instance of the component type $\mathsf{SubseaProdSystem}$, which defines two configurable features xTs_{dc} and $\mathsf{semApps}_{dc}$ (See Example 2). Both configurable features are cardinality features and one instance of each exists in the set of configurable parameters of toySps. Both configurable parameters are configured, in Figure 4, by assigning the value one to them. As a result of this configuration one instance of $\mathsf{XmasTree}$ and one instance of SemApp are created. As shown in Figure 4 these instances are named $\mathsf{xt1}$ and $\mathsf{semAppA}$, respectively.

Let $P = (C, \preceq, N, L)$ be a product, and let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be the reference architecture corresponding to P. We use P_C to denote the set of all configurable parameters in the product P. Specifically, $P_C = \bigcup_{c \in C} P_c$. For a component $c \in C$, several instances of each configurable feature in $F_{\mathcal{T}(c)}$ may exist in P_c. Let f be a feature in $F_{\mathcal{T}(c)}$, and let $\rho_c(f)$ denote the set of all configurable parameters $q \in P_c$, such that q is an instance of f. We refer to the cardinality of the set $\rho_c(f)$ as the cardinality of feature f in the component c. Let $ct \in CT$ be a component type, and let $f \in F_{ct}$ be one of its configurable features. We say that f has a fixed cardinality, iff $\forall c, c' \in [\![ct]\!].\ \sharp\rho_c(f) = \sharp\rho_{c'}(f)$. All cardinality features have fixed cardinalities and their cardinalities is always one. Specifically, let ct be a component type, then for every cardinality feature f of ct (i.e., $f \in AC_{ct} \cup DC_{ct} \cup SC_{ct}$), we have $\forall c \in [\![ct]\!].\ \sharp\rho_c(f) = 1$. Moreover, for each configurable feature with an unfixed cardinality, there exists a corresponding cardinality feature in the same component type.

Fully- and partially-configured components and products

Recall from Section 3 that a component is configured by assigning values to its configurable parameters. A component is called *fully-configured* iff all of its configurable parameters are assigned values, and it is called *partially-configured* iff at least one of its configurable parameters is not assigned a value. Similarly, a product is called fully-configured if all of its contained components are fully-configured, and it is partially-configured if at least one of its contained components is partially configured. The set $[\![RA]\!]$ defined above contains both fully- and partially-configured products. Similarly, the set $[\![ct]\!]$ contains both fully- and partially-configured components.

4.3 Product consistency

A central concept in our approach is the consistency of a product with respect to its reference architecture. Intuitively, a product is consistent with its reference architecture, if and only if all the four conditions listed in Figure 11 hold. Definition 5 formally defines the notion of product consistency.

1. Each component in the product is an instance of a component type in the reference architecture.
2. A component of type *ct* contains a subcomponent of type *ct′* only if there is a decomposition relationship between *ct* and *ct′* in the reference architecture.
3. Two components of types *ct* and *ct′* are connected only if there is an association between component types *ct* and *ct′* in the reference architecture.
4. The product satisfies all the constraints defined in the reference architecture.

Fig. 11. Product consistency rules.

Definition 5 (Product consistency). *Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture, and let $P = (C, \preceq, N, L)$ be a fully- or partially-configured product in $[\![RA]\!]$. We say the product P is consistent w.r.t. RA, iff:*

1. $\forall c \in C.\ \mathcal{T}(c) \in CT$,
2. $\forall (c, c') \in \preceq .\ \exists (ct, ct', l, u) \in \prec:\ c \in [\![ct]\!] \wedge c' \in [\![ct']\!]$,
3. $\forall c \in C, \forall ct \in types(c), \forall (ct, ct', l, u) \in \prec .\ l \leq \sharp\{(c, c') \in \preceq | c' \in [\![ct']\!]\} \leq u$,
4. $\forall (c, n, c') \in L.\ \exists n \in N : \mathcal{A}(n) = (ct, l, u, ct', l', u') \wedge c \in [\![ct]\!] \wedge c' \in [\![ct']\!]$,
5. $\forall c \in C, \forall ct \in types(c), \forall n \in N.[(\ \mathcal{A}(n) = (ct, l, u, ct', l', u') \Rightarrow l' \leq \sharp\{(c, n, c') \in L | c' \in [\![ct']\!]\} \leq u') \wedge (\mathcal{A}(n) = (ct'', l'', u'', ct, l, u) \Rightarrow l'' \leq \sharp\{(c'', n, c) \in L | c'' \in [\![ct'']\!]\}| \leq u'')]$,
6. $C \models \Phi\ (\equiv \forall \phi \in \Phi.\ C \models \phi)$.

The first item in Definition 5 provides the formal specification of the first condition in Figure 11. The second and the third items in the definition address the second consistency rule. The third item, in particular, ensures that the numbers of subcomponents of each component are consistent with the multiplicities of the corresponding decompositions. The next two items in the definition address the third consistency rule about associations. The fifth item in the definition specifies the consistency with respect to the multiplicities of association relations. Finally, the last item in Definition 5 specifies the last consistency rule in Figure 11.

Example 6. The product in Figure 4 is consistent w.r.t. the reference architecture represented by the SimPL model in Figure 3. In particular, the last item in Definition 5 holds for the product in Figure 4 as the values of attributes pinIndex and ebIndex in semAs1 and the value of attribute eBoards in semA satisfy the constraints ϕ_{PinRange} and $\phi_{\mathsf{BoardIndRange}}$ (i.e., the OCL constraints in Section 3.1).

Based on the notion of product consistency we define the consistency of a reference architecture (e.g., representing a SimPL model) as follows:

Definition 6 (Consistent reference architecture). *A reference architecture RA is consistent iff there is a fully-configured product $P \in [\![RA]\!]$, such that P is consistent w.r.t. RA.*

In our context, a SimPL model represents a consistent reference architecture if it conforms to the UML metamodel and the SimPL profile defined in [8], and its set of constraints are consistent. There is a large body of research dealing with UML/OCL consistency [13, 14, 25, 32]. In our approach to configuration, we assume the input reference architecture is consistent (i.e., its consistency is guaranteed using one of the approaches listed above). Checking the consistency of a reference architecture is, however, out of the scope of this paper.

4.4 Syntax and semantics of constraints

As noted in Definition 1, a reference architecture incorporates a set of constraints. In this section, we provide a syntax for such constraints and formally define the satisfiability of such constraints in the context of product configuration. In our work, we focus on a subset of OCL to specify our constraints. This subset is sufficiently expressive to capture all the constraints in our industrial case study. Specifically, our constraint language captures all arithmetic and first order logic formulas described over primitive literals (e.g., numbers), and over sets or individual components and their attributes. By basing our constraint language on first order logic, we can efficiently encode the reference architecture constraints in an input program for constraint solvers (e.g., [15, 4]).

4.4.1 A grammar for boolean formulas

Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture. Each member ϕ of the set Φ of constraints is a tuple (ct, φ), where $ct \in CT$ is a component type denoting the context of the constraint ϕ, and φ is a boolean expression over the set F_{CT} of the configurable features defined in RA. The boolean expression φ can only be evaluated for instances of ct, and must be true for all of them. Each constraint ϕ, therefore, equivalent to the formula $\forall c \in [\![ct]\!].\varphi$.

A simplified grammar for the language of boolean expressions is given in Figure 12. This grammar is defined based on the basic OCL operators that we use

```
bool_expr   ::= bool_term (OR bool_term)*;
bool_term   ::= bool_factor (AND bool_factor)*;
bool_factor ::= bool_literal | qualified_name |
                '(' bool_expr ')' | rel_expr | NOT bool_factor |
                FA '(' qualified_name ',' bool_expr ')' |
                EX '(' qualified_name ',' bool_expr ')' ;

rel_expr    ::= num_expr (GT | LT | GEQ | LEQ | EQ | NEQ) num_expr;

num_expr    ::= num_term ((PLUS | MINUS) num_term)*;
num_term    ::= num_factor ((MUL | DIV) num_factor)*;
num_factor  ::= num_literal | qualified_name |
                '(' num_expr ')' | NEG num_factor;
```

Fig. 12. A simplified grammar of boolean formulas.

in specifying constraints in the SimPL methodology. These operators include, for all, exists, arithmetic, relational and logical operators. In Figure 12, FA represents the universal quantifier, which maps to OCL forAll operator. Similarly, EX represents the existential quantifier, which maps to OCL exists operator.

In the grammar given in Figure 12, a qualified name (i.e., qualified_name) represents a typed variable (i.e., a configurable feature[3]), and may represent an individual item or a collection of items. For example, in the OCL expressions in Section 3.1, the qualified name pinIndex represents a single item of type integer, and the qualified name sem.eBoards represents a collection of items, where each item is typed by the enumeration ElecBoard. In the formula $\forall c \in [\![ct]\!].\varphi$, all qualified names in φ are of the form $c.a_1.a_2....a_n$[4], where a_1 refers to an attribute of c, and a_i (for $1 < i \leq n$) refers to an attribute of a_{i-1}. In addition, a_n can be "$size()$". In this case, the qualified name $c.a_1...a_{n-1}.size()$ represents the size of the collection $c.a_1...a_{n-1}$.

Qualified names together with literals and operators are used to create numerical, relational, and boolean expressions. Qualified names of numerical types (i.e., integer or a user defined enumeration) form one type of numerical factors and are used in creating relational expressions. Qualified names of type boolean form one type of boolean factors. Qualified names representing collections of items can be combined with set quantifiers (i.e., for all and exists) to form another group of boolean factors.

[3] Note that in our approach, we distinguish between configurable features and the features (e.g., attributes) that are evaluated at run-time. We call the latter non-configurable features. In our approach, non-configurable features may exist in the reference architecture, but they cannot be used for defining constraints since their values can change at run-time. As non-configurable features cannot be used in expressing constraints, they are excluded from our formal specification.

[4] Note that, whenever needed in the OCL expressions used in this paper, we have used -> is instead of *dot*, to conform to OCL syntax.

4.4.2 Constraint satisfiability semantics

Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture, and let $P = (C, \preceq, N, L)$ be a product consistent w.r.t. RA. The last item in Definition 5 requires that the components in the set C and the values of their configurable parameters satisfy all the constraints defined in Φ. In this section, we precisely define the semantics of constraint satisfiability based on the elements of P and RA.

Let $\phi = \forall c \in [\![ct]\!].\varphi$ be a constraint in Φ, and let $q_1, ..., q_n$ denote all the qualified names in the boolean expression φ (or $\varphi(q_1, ..., q_n)$ to be more precise). Recall that each q_i is of the form $c.a_1.a_2....a_k$. For each instance c of ct, each qualified name q_i in φ specifies a particular subset of P_C, which we denote as $\mathcal{R}(q_i, c)$[5]. Let $\mathcal{L}(q_i, c)$ be a list representation of the elements in $\mathcal{R}(q_i, c)$. Using this definition, we rewrite the boolean expression $\varphi(q_1, ..., q_n)$ for each component $c \in [\![ct]\!]$ and denote it as $\mathcal{R}(\varphi, c)$, and we have:

$$\mathcal{R}(\varphi, c) \equiv \mathcal{R}(\varphi(q_1, ..., q_n), c) \equiv \varphi(\mathcal{L}(q_1, c), ..., \mathcal{L}(q_n, c)).$$

Note that $\mathcal{R}(\varphi, c)$ is a boolean expression over the configurable parameters in P_C. Let $p_1, ..., p_m$ denote all the configurable parameters in the boolean expression $\mathcal{R}(\varphi, c)$. We say that $c \models \varphi$ (i.e., c satisfies φ) iff there exist values $v_1, ..., v_m$ for parameters $p_1, ..., p_m$ that evaluate the boolean expression $\mathcal{R}(\varphi, c)$ to true. To denote it mathematically, we write:

$$c \models \varphi \;\Leftrightarrow\; \exists v_1, ..., v_m : \mathcal{R}(\varphi, c)(v_1, ..., v_m) = true. \tag{1}$$

Definition 7 (Constraint satisfiability). *Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture, and $P = (C, \preceq, N, L)$ be a product in $[\![RA]\!]$. For each constraint $\phi = (ct, \varphi)$ in the set Φ, we say that C satisfies ϕ and denote it as $C \models \phi$ iff:*

$$\forall c \in [\![ct]\!].\; c \models \varphi. \tag{2}$$

Example 7. The OCL constraints in Section 3.1 are written in the context of component type ElectronicConnection. In Figure 4, the only instance of ElectronicConnection is the component semAs1. Therefore, in order to check the consistency of the product in Figure 4, the constraints ϕ_{PinRange} and $\phi_{\mathsf{BoardIndRange}}$ should only be evaluated for semAs1. To do so, we rewrite these constraints for component semAs1, resulting in two boolean expressions in terms of the configurable parameters of the components in Figure 4. The following shows these two boolean expressions, both of which evaluate to true.

```
semAs1.pinIndex >= 0 and
semA.eBoards[semAs1.ebIndex+1].numOfPins > semAs1.pinIndex

semAs1.ebIndex >= 0 and semAs1.ebIndex < semA.eBoards_ac
```

[5] We apply name resolution rules, typically found in programming languages, for resolving qualified names with dot-notations, to calculate the set $\mathcal{R}(q_i, c)$.

As mentioned in Section 4.2, products may be partially configured. Fully-configured products can be created from partially-configured products by configuring all their unconfigured parameters. In addition, new components may be needed to create a fully-configured product from a partially-configured product. According to Definition 7, when assessing the consistency of a partially-configured product, we evaluate constraints only for its existing components, not the ones that may be added later.

5 The configuration process

In this section, the process of creating an individual product from a reference architecture is explained. The configuration process is a stepwise process, where in each step a value is assigned to a configurable parameter. We call each configuration step a *value-assignment* step. As a result of assigning a value to a configurable parameter, new components or new configurable parameters may be created. Here, we first explain the process of configuring a single component then we explain the overall product configuration process. We define both processes using the notations introduced in Definitions 1-4.

5.1 Component configuration

Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture, $P = (C, \preceq, N, L)$ be a partially-configured product of RA, and $c \in C$ be an unconfigured or partially-configured component. Suppose c is an instance of the component type $ct \in CT$, and has been created in an earlier step during the configuration of product P. When instantiating a component type some of its configurable features are instantiated immediately and some of them are instantiated later during the configuration process, when the information required for instantiating them is provided. More specifically, a configurable feature is instantiated immediately iff it has a fixed cardinality. Examples of such configurable features are cardinality features and configurable attributes that have fixed multiplicities of the form $k..k$. For example, one instance of each cardinality feature defined by the component type ct is created and added to P_c when the component c is created. Configurable features that have unfixed cardinalities are instantiated only after their corresponding cardinality features are configured.

In this section, we explain how a single component is configured and what are the consequences of each configuration step. To do so, we assume that at each configuration step a value is assigned to a configurable parameter and explain the details for each kind of configurable parameter separately. In the following, $c \in C$ is a component in the context of product P, $q \in P_c$ is the configurable parameter to be configured, and ct denotes the concrete type of the component c (i.e., $ct = \mathcal{T}(c)$).

5.1.1 Configurable attribute (members of AF_{ct}).

Let f be a feature denoting a configurable attribute in AF_{ct} and let q be an instance of f. The configurable parameter q is configured by assigning to it a value from the type of the the configurable attribute f (i.e., $type(f)$).

5.1.2 Configurable attribute cardinality (members of AC_{ct}). Let f be a feature denoting a configurable attribute cardinality in AC_{ct} and let q be an instance of f. The configurable parameter q is configured by assigning to it a valid integer value k. As a result of this value-assignment step, k configurable parameters – each an instance of the configurable attribute $atr(f)$ – are added to the set P_c. These configurable parameters are to be configured in the later steps of configuration.

5.1.3 Configurable decomposition cardinality (members of DC_{ct}). Let f be a feature denoting a configurable decomposition cardinality in DC_{ct} and let q be an instance of f. The configurable parameter q is configured by assigning to it a valid integer value k. Depending on whether $chl(f)$ (i.e., the component type denoting the target of the corresponding decomposition relationship) has some subtypes or not, one of the following two cases is followed as a result of this configuration step:

1. $chl(f)$ **has no subtypes.** In this case, k new components, each an instance of $chl(f)$, are created and added to the set C. Let $c_1, ...c_k$ be the newly created components. For each component c_i we add the tuple (c, c_i) to the set $\preceq$. The newly created components are to be configured in the later steps of configuration.
2. $chl(f)$ **has a number of subtypes.** In this case, k new configurable parameters are added to the set P_c of configurable parameters. Each configurable parameter represents a configurable type and is an instance of a configurable feature $f' \in DT_{ct}$, where $chlT(f') = chl(f)$. Each newly created configurable parameter is to be configured in a later step of configuration.

5.1.4 Configurable association cardinality (members of SC_{ct}). Let f be a feature denoting a configurable association cardinality in SC_{ct} and let q be an instance of f. The configurable parameter q is configured by assigning to it a valid integer value k. As a result of this value-assignment step, k new configurable parameters are added to the set P_c. Each of the configurable parameters is an instance of a configurable topology $f' \in TF_{ct}$, where $acsT(f') = acs(f)$. Each of the newly created configurable parameters is to be configured in a later step of configuration.

5.1.5 Configurable type (members of DT_{ct}). Let f be a feature denoting a configurable type in DT_{ct} and let q be an instance of f. The configurable parameter q is configured by assigning to it the name of a component type $ct \in subtypes(chlT(f))$. As a consequence of this configuration step a new instance of ct is created and added to the set C. Let c' be the newly created component. We add the tuple (c, c') to the set $\preceq$ as a result of this configuration step. This newly created component is to be configured in the later steps of configuration.

5.1.6 Configurable topology (members of $\boldsymbol{TF_{ct}}$).

Let f be a feature denoting a configurable topology in TF_{ct} and let q be an instance of f. The configurable parameter q is configured by assigning to it the name of a component already existing in the set C. Suppose c' is the chosen component. As a result of this configuration step, we add the tuple $(c, ascT(f), c')$ to the set L.

Table 3. Summary of the configuration of each kind of configurable parameter.

Feature kind	**Configuration decision**	**Change of the state of P in each iteration**
Configurable attribute ($f \in AF_{ct}$)	$v(q) \in type(f)$	–
Configurable attribute cardinality ($f \in AC_{ct}$)	$v(q) = k \in \mathbb{N}_0$	$P_c^{i+1} = P_c^i \cup \{q_1, .., q_k\}$, $q_j \in \rho_c(atr(f))$

Configurable decomposition cardinality ($f \in DC_{ct}$)	$v(q) = k \in \mathbb{N}_0$	if $subtypes(chl(f)) = \emptyset$: $C^{i+1} = C^i \cup \{c_1, .., c_k\}$, $c_j \in [\![chl(f)]\!]$, $\preceq^{i+1} = \preceq^i \cup \bigcup_{j=1..k}\{(c, c_j)\}$
		
		 else ($\exists f' \in DT_{ct}$: $chl(f) = chlT(f')$): $P_c^{i+1} = P_c^i \cup \{q_1, .., q_k\}$, $q_j \in \rho_c(f')$

Configurable association cardinality ($f \in SC_{ct}$)	$v(q) = k \in \mathbb{N}_0$	$P_c^{i+1} = P_c^i \cup \{q_1, .., q_k\}$, $q_j \in \rho_c(f'), f' \in TF_{ct}$, $asc(f) = ascT(f')$
Configurable type ($f \in DT_{ct}$)	$v(q) = ct'$, $ct' \in subtypes(chlT(f))$	$C^{i+1} = C^i \cup \{c'\}$, $c' \in [\![ct']\!]$, $\preceq^{i+1} = \preceq^i \cup \{(c, c')\}$
Configurable topology ($f \in TF_{ct}$)	$v(q) = c' \in C$	$L^{i+1} = L^i \cup \{(c, ascT(f), c')\}$

Table 3 formalizes the details of configuring component c as explained above. In this table, we use ct to denote the type of component c, $q \in P_c$ is the configurable parameter that is configured, $f \in F_{ct}$ is the respective configurable feature, and $v(q)$ denotes the value assigned to q. We use C^i, $\preceq^i$, L^i, and P_c^i to specify the state of product P in the ith step of configuration. We assume these sets remain unchanged as a result of completing the configuration step unless stated otherwise.

5.2 Product configuration

The process of creating a product $P = (C, \preceq, N, L)$ from a reference architecture $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ starts by creating an instance of the topmost component type in the SimPL model of RA. The topmost component type is a

UML class stereotyped by «ICSystem» in the SimPL model and is denoted by $ct_s \in CT$. Each product has only one instance of the topmost component type. We refer to this single instance of ct_s as c_s.

The configuration process starts after the component c_s with its initial list of configurable parameters is created. At this stage, for the product P we have $C = \{c_s\}$, $\preceq = \emptyset$, and $L = \emptyset$. Most of the configurable parameters in P_{c_s} are of type cardinality at this stage, and so, in the first steps of configuration these configurable cardinalities are configured. Configuration of such parameters eventually results in the creation of new components. The configuration process then proceeds with configuring these new components. The overall configuration process is given in Algorithm 1. This algorithm and the rest of algorithms in this paper follow the pseudocode convention given in [17].

Algorithm 1 CONFIGURATION

```
Input: a reference architecture $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$
Output: a product specification $P = (C, \preceq, N, L)$
  ▷ Initialization
1 $c_s \leftarrow$ createNew($ct_s$)
2 $C \leftarrow \{c_s\}$
3 while there are unconfigured parameters in $P_C$ do
4     read($i$)
5     read($P_C[i]$)
6     applyConfigDecision($P_C[i], P$)  ▷ May result in the creation of new configurable parameters, new components, or new relations.
7 return $P$
```

In each iteration of the while-loop in Algorithm 1, one user-selected configurable parameter is configured by assigning a user-provided value to it. The user selects the desired parameter in line 4, and assigns a value to it in line 5. The algorithm terminates when all the configurable parameters are assigned a value. The output in this case is a complete product specification. In practice, the algorithm may be terminated at the end of each configuration iteration if the user decides to do so. In this case, the output is a partial product specification. The function createNew, in the configuration algorithm, creates a new instance of the input component type; and the function ApplyConfigDecision applies the configuration decision and updates the product specification as specified in Section 5.1 and summarized in Table 3. An algorithm implementing ApplyConfigDecision is indeed a switch-case statement, which can be derived from Table 3. For each kind of configurable feature (i.e., each row of Table 3), we have a case statement in the algorithm. In each case statement, the action is specified by the third column of the table.

6 Semi-automated configuration

The semi-automated configuration approach in Figure 1 performs three main activities: (1) ensuring the consistency of the products by validating configuration decisions, (2) automatically making some of the configuration decisions, and (3) guiding the user throughout the configuration process. We refer to these activities as *configuration functionalities*. Central to the implementation of the configuration functionalities is the notion of *valid domains*. Given a partially-configured product, the valid domain of a configurable parameter specifies the set of all values that can be assigned to that configurable parameter without resulting in any inconsistencies in the product. After each individual value assignment, we need to update the valid domains accordingly. We use constraint propagation over finite domains to recompute the valid domains at the end of each configuration iteration. To do so, we cast the configuration problem as a constraint satisfaction problem over finite domains.

In the configuration process presented in Section 5, we have neither addressed the computation of the valid domains nor addressed the configuration functionalities mentioned above. In this section, we explain how constraint propagation techniques are intertwined with the configuration process presented in Section 5 to fill in this gap. In the remainder of this section, we first introduce the finite-domains constraint program and describe some basic operations on it that help computing the valid domains. We specify the mapping from the configuration problem to the constraint program. Then we explain how the configuration functionalities can be implemented using the valid domains. Finally, we present a modified version of the configuration process of Section 5. In this modified version, calculation of the valid domains and implementation of the configuration functionalities are included. Using the definitions and algorithms given in this section, we present and prove the main properties of our semi-automated configuration approach in Section 7.

6.1 A finite-domains constraint program

A finite-domains constraint program *cp* is a quadruple $(X, \mathcal{D}, V, \Psi)$, where X is a set of variables $x_1, ..., x_n$, $\mathcal{D}$ is a set of finite domains $D_1, ..., D_n$, V is a set of values for the variables in X, and Ψ is a set of constraints over the variables in X. Variables in X get their values from the *finite domains* $D_1, ..., D_n$. A finite domain D_i is a finite collection of tags that can be mapped to unique integers. Let ψ be a constraint in Ψ, and let $x_1, ..., x_m$ be the variables in ψ. We say that $\psi(v_1, ..., v_m)$ holds iff the values $v_1, ..., v_m$ in $D_1, ..., D_m$ evaluate the constraint ψ to true.

Several operations can be defined on a constraint program, including consistency checking and constraint solving. For the purpose of consistent product configuration, we are interested in two operations defined in [29]: checking *domain satisfiability* and computing *reduced domains*, which are defined below. The following definitions are borrowed from [29] but slightly modified to match our definition of constraint program.

Definition 8 (Domain consistency). *A constraint ψ is domain-consistent w.r.t. $D_1, ..., D_n$ if, for each variable x_i and value $v_i \in D_i$, there exist values $v_1, ..., v_{i-1}, v_{i+1}, ..., v_n$ in $D_1, ..., D_{i-1}, D_{i+1}, ..., D_n$ such that $\psi(v_1, ..., v_n)$ holds. A constraint program $cp = (X, \mathcal{D}', V, \Psi)$ is domain-consistent w.r.t. $D_1, ..., D_n$ if any constraint ψ in Ψ is domain-consistent w.r.t. $D_1, ..., D_n$.*

Definition 9 (Reduced domains). *The reduced domains of a constraint program $cp = (X, \mathcal{D}, V, \Psi)$ are the largest domains $D''_1, ..., D''_n$ such that cp is domain-consistent w.r.t. $D''_1, ..., D''_n$, i.e., for all domains $D'_1, ..., D'_n$ such that cp is domain-consistent w.r.t. $D'_1, ..., D'_n$, we have $D'_1 \subseteq D''_1 \ \wedge ... \ \wedge \ D'_n \subseteq D''_n$.*

As noted in [29], the reduced domains of a constraint program *cp* exist and are unique and all the solutions of *cp* are in its reduced domains. Domain consistency is thus a sound approximation of consistency.

Definition 10 (Domain satisfiability). *A constraint program is domain-satisfiable iff none of its reduced domains is empty.*

Constraint propagation is a technique for calculating the reduced domains of a constraint program. Given a set of constraints and an initial set of finite domains for the variables in those constraints, a constraint propagation algorithm iterates over the set of constraints and for each constraint prunes the domains of the involved variables by removing inconsistent values from them. The algorithm iterates over the set of constraints until no more pruning is possible. Algorithm 2[6] shows the constraint propagation algorithm [40, 10].

Algorithm 2 Constraint Propagation Algorithm (AC-3)

```
Input: $\sigma$ a set of constraints, $D$ a set of finite domains
Output: $D'$ arc-consistent domains
1 $queue \leftarrow \sigma$
2 while $queue \neq \emptyset$ do
3     $c \leftarrow \text{dequeue}(queue)$
4     $D' \leftarrow \text{reduce}(c, D)$ ▷ Domain reduction
5     if $D' \neq D$ then
6         $queue \leftarrow queue \cup \{c' \in \sigma |\ \textsc{Vars}(c') \cap \textsc{Vars}(c) \neq \emptyset\}$
7         $D \leftarrow D'$
8 return $D'$
```

The constraint propagation algorithm is monotonic and terminates after a finite number of iterations. The algorithm is monotonic because, during each iteration, the size of each domain either decreases or remains unchanged. The algorithm terminates after a finite number of iterations because there are a finite number of input domains and a finite number of elements in each input domain.

[6] AC-3 is the basic and simplest Arc-consistency algorithm used for constraint propagation. More advanced and more efficient versions of this algorithms exist (i.e., AC-4 through AC-7), which may, in practice, be used for calculating the reduced domains. A comparison of these variations can be found in [57].

6.2 Consistent configuration through constraint programming

A product can be represented by a constraint program. In this section, we first present a mapping specifying how a constraint program can be created from a product specification. Then, we explain the changes that take place in the state of the constraint program after each configuration step. Finally, we provide an analysis of the time complexity of creating constraint programs from products.

6.2.1 *Mapping a product to a constraint program.*

Let $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ be a reference architecture, and let the partially-configured product $P = (C, \preceq, N, L)$ be a member of $[\![RA]\!]$. To create the constraint program $cp = (X, \mathcal{D}, V, \Psi)$ for the product P, we add a variable to the set X for each configurable parameter of each component in C. If the configurable parameter is already assigned a value, we add the configuration value to the set V, otherwise we add the symbol ϵ to V to denote that the respective configurable parameter is not yet configured. For a configurable parameter $q \in P_C$ being an instance of a configurable feature $f \in F_{CT}$, a finite domain D_q representing the initial domain of the configurable parameter q is computed using the configurable feature f and is added to the set $\mathcal{D}$. For each constraint $\phi = (ct, \varphi) \in \Phi$, and each component $c \in [\![ct]\!]$, we add a constraint $\mathcal{R}_X(\varphi, c)$ to Ψ. Such a constraint is the same as $\mathcal{R}(\varphi, c)$ defined in Section 4.4.2, except that configurable parameters are replaced by their equivalent variables in X.

Table 4 summarizes the mappings discussed above. For each element in a constraint program, the related concepts in the product specification or the reference architecture are listed in Table 4. In this table the column named Origin specifies whether the concepts belong to the product specification or the reference architecture.

Table 4. Summary of the mapping between product configuration concepts and the elements in a constraint program.

Constraint program element	**Product configuration concept**	
	Related concepts	**Origin**
Variables	Configurable parameters	Product specification
Domains	Configurable features	Reference architecture
Values	Configuration decisions	User-provided data (Product specification)
Constraints	Constraints	Reference architecture

In the following, we further elaborate how the domains in $\mathcal{D}$ are initialized for each type of configurable features. Let $f \in F_{CT}$ be a configurable feature defined in RA, let the configurable parameter $q \in P_C$ be an instance of f, and let $x \in X$ be the variable representing the configurable parameter q.

***f* is a configurable attribute.** In this case, the initial domain of variable x is specified by the type of the attribute represented by the feature f. The type of an attribute, in our context, can be boolean (i.e., $\{T, F\}$), an integer interval (i.e., $(lower..upper)$), or an enumeration (i.e., $\{t_1, ..., t_n\}$). In all cases the initial domain of x will be a finite set of tags.

f is a configurable cardinality. In this case, the initial domain of variable x is specified by an integer interval (i.e., $(lower..upper)$). Such an integer interval is extracted from the multiplicities in the reference architecture RA. If the feature f is a configurable decomposition cardinality related to the decomposition $(ct, ct', l, u) \in \prec$, then the initial domain of variable x is the integer interval indicated by $l..u$. The initial domains for configurable attribute cardinalities and configurable association cardinalities can be defined similarly.

f is a configurable type. In this case, the initial domain of variable x is specified by a finite set of tags, where each tag is the name of a subtype of $chlT(f)$.

f is a configurable topology. In this case, the initial domain of variable x is specified by a finite set of tags, where each tag is the name of a component in C. More specifically, if f is introduced by association $\mathcal{A}(n) = (ct, l, u, ct', l', u')$, then the initial domain of x would contain all members of C that denote a component of type ct' (i.e., $D_x = [\![ct']\!]$).

The initial domains are pruned using the constraint propagation algorithm to calculate the reduced domains of the variables in X. For a constraint program cp representing a product P, the reduced domains of its variables exist and form the valid domains of the configurable parameters in the product P. According to Definition 10, if the constraint program is domain-satisfiable, then all of such valid domains are non-empty.

6.2.2 *Changes in the state of the constraint program.*

Suppose $P = (C, \preceq, N, L)$ is a partially-configured product under configuration, $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$ is the related reference architecture, and $cp = (X, \mathcal{D}, V, \Psi)$ is the constraint program created for P. We use $X^i, \mathcal{D}^i, V^i$ and Ψ^i to denote the state of the constraint program cp in the ith step of configuration. In the following we explain how the state of the constraint program cp changes during the ith step of configuration as a result of assigning a valid value v_j to variable x_j representing an instance of the configurable feature f.

General rule. As a general consequence of the aforementioned value-assignment step, the jth element in the set V^i will be replaced by v_j in V^{i+1}.

f is a cardinality feature. As discussed in Section 5.1, as a result of configuring a configurable parameter of type cardinality new configurable parameters may be added to the set of configurable parameters of the related component (See rows 2-4 in Table 3). For each new configurable parameter a new variable is added to the constraint program. In other words, if in the ith step of configuration a parameter of type cardinality is configured by assigning to it the integer value v_j, then X^{i+1} will have v_j more elements than X^i. Similarly, V^{i+1} will have v_j more elements than V^i, and all of its additional elements will be set to ϵ.

In certain cases (i.e., Section 5.1.3), configuring a configurable parameter of type cardinality may result in the creation of new components. This is also the case when an instance of a configurable type is configured (Section 5.1.5).

f is a configurable type. As a result of a value-assignment step new components may be added to C. This is the case if, for example, the configured parameter represents a configurable type (See the fifth row in Table 3). As a result of adding a new component c to C, both a number of variables and a number of constraints must be added to the constraint program *cp*. Assuming component c has k configurable parameters at the time of its instantiation, we add k new elements to the sets of variables and values (i.e., X^{i+1} and V^{i+1} will each have k elements more than X^i and V^i, respectively). In addition, for each constraint $\phi = (ct, \varphi) \in \Phi$ where $ct = \mathcal{T}(c)$ (i.e., ϕ is a constraint defined in the context of the type of the component c), we add a constraint $\mathcal{R}_X(\varphi, c)$ to the set Ψ. In other words, if l is the number of constraints defined in the context of $\mathcal{T}(c)$, then Ψ^{i+1} will have l elements more than Ψ^i as a result of this configuration step.

6.2.3 *Time complexity of creating the constraint program*

Lemma 1. *Time complexity of creating a constraint program $cp = (X, \mathcal{D}, V, \Psi)$ for a product $P = (C, \preceq, N, L)$ is linear with the number of configurable parameters in the product P.*

Proof. The complexity of creating a constraint program is equal to the sum of the complexities of creating variables, their initial domains, and the constraints. A variable can be created in constant time from its corresponding configurable parameter. Similarly, each initial domain can be specified in constant time. The time complexity for creating variables and their corresponding initial domains is, therefore, equal to $O(Q)$, where Q is the total number of configurable parameters in the product.

As discussed earlier in this section, a number of constraints are added to the constraint program each time a new component c is created. Each constraint is created by applying the transformation $\mathcal{R}_X$ introduced at the beginning of Section 6.2. This transformation has a complexity proportional to Q (i.e., $O(Q)$). As a result, the total complexity of creating the constraint program is $O(Q)$.

6.3 Implementing the configuration functionalities

As mentioned earlier, in our semi-automated configuration approach, we propose three configuration functionalities. In this section, we describe how the two first functionalities can be implemented having the valid domains. Valid domains can as well be provided to the users as a form of configuration guidance, therefore implementing the third functionality. In the following, let $P = (C, \preceq, N, L)$ be a partially-configured product, and let $cp = (X, \mathcal{D}, V, \Psi)$ be the constraint program created for product P.

Validation of the configuration decisions. Suppose that in the ith step of configuration of the product P, the user has assigned the value v to a configurable parameter $q \in P_C$. Let $x_i \in X$ be the variable representing the configurable parameter q in the constraint program *cp*. We use the reduced domain $D_i \in \mathcal{D}$ to validate the value v. Specifically, the user-provided value v represents a valid configuration decision if $v \in D_i$.

Automated decision making. At the end of each configuration iteration, we use the valid domains (i.e., $D_i \in \mathcal{D}$) to infer some of the configuration decisions automatically. To do so, we check the size of all of the valid domains. A valid domain of size one indicates that, given all other decisions, there is only one way to configure the corresponding configurable parameter. Therefore, for each valid domain D_i containing only one value v, we automatically configure the configurable parameter corresponding to the variable x_i by assigning the value v to it. Algorithm 3 shows the value inference algorithm.

Algorithm 3 INFERVALUES

Input: a constraint program $cp = (X, \mathcal{D}, V, \Psi)$, and a product $P = (C, \preceq, N, L)$, and a list of decisions *dList*
Output: updated *cp* and P

```
1  decisions : empty list
2  while there are singleton domains in $\mathcal{D}$ do
3    for each singleton domain $D_i = \{v\}$ do
4      $x_i \leftarrow v$
5      APPEND($decisions, (x_i, v_i)$)
6      APPLYCONFIGDECISION($x_i, v, P, cp$)
7      UPDATEVALIDDOMAINS($P, cp$)
8      if some domains are empty in $cp$ then
9        rollBack($decisions, p, cp$)
10       return $False$
11 APPEND($dList, decisions$)
12 return $True$
```

Inferring the value of a configurable parameter has the same consequences as that of configuring such a configurable parameter. These consequences (discussed in Sections 5.1.2-5.1.4) are of particular importance in the case of configurable cardinalities and configurable types. Specifically, inferring the value of a configurable cardinality or a configurable type results in creating a number of new configurable parameters or a number of new components and constraints in the product under configuration. As a result of this change in the product specification, the constraint program is updated accordingly. Furthermore, in every value-inference round, after assigning a value to a variable, we may need to prune valid domains of other variables. That is why after each round of value inference, we recompute the valid domains of the remaining variables in line 7. This may result in some empty valid domains. Once we detect an empty valid domain, we roll back all the inferred values (line 9), and return false, indicating that the value-inference was not successful. Otherwise, we iteratively infer values for all the singleton valid domains and return true in the end.

6.4 Consistent product configuration

In this section, we present a new configuration algorithm that modifies the configuration algorithm in Section 5 (i.e., Algorithm 1) by taking into account the consistency aspects. In particular, we specify how configuration functionalities are invoked during the configuration process to ensure the consistency of the final product.

Algorithm 4 ConsistentConfiguration

```
Input: a reference architecture $RA = (CT, \prec, \mathcal{G}, N, \mathcal{A}, \Phi)$
Output: a consistent product specification $P = (C, \preceq, N, L)$
   ▷ Initialization
 1 $c_s \leftarrow$ createNew($ct_s$)
 2 $C \leftarrow \{c_s\}$
 3 $cp \leftarrow$ createConstraintProgram($RA, P$)
 4 $dList$ : $empty\ list\ of\ (variable,\ value)\ pairs$
 5 while there are unconfigured parameters in $P_C$ do
 6    read($i$)   ▷ $i$ indicates the parameter that will be configured
 7    do
 8       read($tmp$)   ▷ assigned value
 9       $readNext \leftarrow True$
10       if isValid($tmp, D_{P_C[i]}$) then
11          $readNext \leftarrow False$
12          append($dList, (i, tmp)$)
13          applyConfigDecision($P_C[i], tmp, P, cp$)
14          updateValidDomains($P_C[i], tmp, cp$)
15          if some domains are empty in $cp$ or not inferValues($cp, P, dList$) then
16             $readNext \leftarrow True$
17             $i \leftarrow$backtrack($dList, p, cp$)
18    while $readNext$
19 return $P$
```

Algorithm 4 is our consistency-oriented configuration algorithm. This algorithm has some additional steps compared to Algorithm 1. After initializing the product, in lines 1 and 2 of the ConsistentConfiguration algorithm, an initial constraint program is created in line 3. The createConstraintProgram function invoked in line 3 performs a first round of constraint propagation to provide an initial version of the valid domains for the initial list of the configurable parameters. The constraint program is used in each configuration iteration (lines 5-18) to provide the configuration functionalities and to ensure the consistency of the configured product[7].

In each configuration iteration, instead of immediately assigning the user-provided value to the selected configurable parameter, we first validate the decision by invoking isValid in line 10. If the decision is valid (i.e., the value is within the valid domain of the chosen configurable parameter), the algorithm proceeds by applying the decision in line 13. To do so, applyConfigDecision is called, which applies the configuration decision and, for each type of configurable features, updates the product as discussed in Section 5.1. In addition, the constraint program will be updated accordingly by adding to it new variables, their initial domains, and new constraints as discussed in Section 6.2. After the configuration decision is applied and the constraint program is updated, the valid domains of all the variables need to be recomputed. This is done in line 14 by a call to updateValidDomains. Finally, at the end of each configuration iteration the function inferValues is invoked to automatically make configuration decisions. This is done in line 15 of the algorithm.

Algorithm 5 backtrack

```
Input: a list of decisions $dList$, a product specification $p$, and a constraint program $cp$
Output: $i$ index of variable $v_i$ in $cp$ that must be configured next
    ▷ Initialization
 1 do
 2     $continue \leftarrow False$
 3     $(i, tmp) \leftarrow dList.last()$
 4     $dList.remove((i, tmp))$
 5     rollBack$((i, tmp), p, cp)$
 6     remove $tmp$ from $D_{P_C[i]}$
 7     if $D_{P_C[i]}$ is empty then
 8         $D_{P_C[i]} \leftarrow$ original valid domain of $v_i$
 9         $continue \leftarrow True$
10 while $continue$ and $dList \neq \emptyset$
11 if $continue$ then
12     return $-1$  ▷ Exception: the input reference architecture is inconsistent
13 return $i$
```

After updating the valid domains in line 14, some of the valid domains in *cp* may become empty. Alternatively, a value inference step (i.e., the inferValues routine) may fail because it yields some empty valid domains. In either case, we have to backtrack the decisions that result in an empty valid domain. In line 15, we check these two possibilities. If it turns out that some valid domains are empty, we call our backtracking algorithm (Algorithm 5) in line 17. Let i be

[7] Note that the constraint program created in line 4, and updated in lines 14 and 15 contains only variables representing the instantiated configurable parameters and the constraints over them.

a variable and tmp be a value assigned to i, and let (i, tmp) be the last tuple in $dList$, i.e., (i, tmp) represents the last decision. In the backtracking algorithm, we remove (i, tmp) from $dList$, and remove the value tmp from the valid domain of i. If this results in an empty valid domain for i, we set the valid domain of i back to the original valid domain, and keep backtracking more. Otherwise, we stop backtracking and ask the user to choose a new value for i different from tmp.

Recomputing the valid domains. As mentioned above, in each configuration iteration, after the configuration decision is applied and the constraint program is updated, the valid domains are recomputed by a call to the function UPDATEVALIDDOMAINS.

If the value-assignment step has not resulted in the creation of any new components, then the valid domains are recomputed by propagating the constraint $\psi : x = tmp$ throughout the constraint program. To do so, we can reuse a previous constraint propagation session and propagate the new constraint by simply inserting it into the queue and executing the while loop of the constraint propagation algorithm (Algorithm 2). We refer to this as on-the-fly constraint propagation, which is guaranteed to preserve the monotonicity of the constraint propagation algorithm. This is due to the fact that the constraint propagation algorithm does not need to know about all the constraints and all the variables a priori. Note that, in this case, the queue is empty before inserting the constraint ψ, but constraints may be added to the queue, in line 6 of Algorithm 2, as a result of the propagation of constraint ψ.

If the value-assignment step has resulted in the creation of some new components, we cannot use the previous constraint propagation session and the on-the-fly constraint propagation to recompute the valid domains. The reason is that, as a result of the creation of new components, new items might have been added to the valid domains of some configurable parameters, and in particular, configurable parameters of type topology. This growth of the valid domains violates the monotonicity of the constraint propagation algorithm, which is necessary for the algorithm to work correctly. Therefore, in this case, instead of using the on-the-fly constraint propagation, we have to start a new constraint propagation session from scratch. In other words, we have to start with a queue containing all the constraints, not only ψ, and pruning the domains until the queue is empty.

7 Properties of the semi-automated configuration

In this section, we characterize the formal properties of our configuration algorithm. In particular, we show that our configuration algorithm always terminates in finite time and generates a fully configured and consistent product. We start by presenting and proving two lemmas that are used in the proof of the formal properties.

Lemma 2 (Finite number of configurable parameters). *Given a reference architecture with no cyclic decomposition relation, value assignment to a configurable parameter does not indefinitely generate new configurable parameters.*

Proof. In our algorithms presented in Section 6, new configurable parameters may be generated as a result of invoking APPLYCONFIGDECISION after value assignment to a configurable parameter of type configurable cardinality or configurable type. To show that value assignments cannot indefinitely generate new configurable parameters, we discuss each value assignment case individually.

Value assignment to configurable parameters of type attribute cardinality or association cardinality: The new configurable parameters generated as a result of this value assignment cannot themselves generate new configurable parameters in the subsequent iterations.

Value assignment to configurable parameter of type decomposition cardinality: This value assignment generates configurable parameters that can lead to generation of new configurable parameters. However, since the decomposition relations in the reference architecture are acyclic, generation of new configurable variables cannot continue indefinitely and has to stop at some point.

Value assignment to configurable parameter of type configurable type: This value assignment can generate only a finite number of configurable parameters of type configurable cardinality. However, as discussed above a configurable parameter of this type does not indefinitely generate new configurable parameters.

□

Lemma 2 implies that the total number of configurable parameters in any possible configuration is finite.

Lemma 3 (Termination of the value-inference algorithm). *Given a consistent reference architecture with no cyclic decomposition relation, the value-inference algorithm (i.e., Algorithm 3) terminates in finite time.*

Proof. Algorithm 3 terminates when there is no "unassigned" variable (configurable parameter) with a singleton valid domain. At each iteration, we assign a value to each configurable parameter with a singleton domain (line 4). We never undo value assignments, unless at line 9, after which we immediately terminate the algorithm. However, a value assignment (line 4) may create new unassigned configurable parameters (at line 6, APPLYCONFIGDECISION). According to Lemma 2, however, value assignments cannot indefinitely generate new configurable parameters. Therefore, the number of configurable parameters ("unassigned" variables) added at line 6 is finite, and Algorithm 3 terminates. □

Theorem 1 (Termination of a configuration iteration). *Given a consistent reference architecture with no cyclic decomposition relation, Algorithm 4 terminates in finite time.*

Proof. To prove that Algorithm 4 is terminating, we note that all its sub-routines are terminating: Algorithm 2 (invoked in line 14 as part of the UPDATEVALID-DOMAIN sub-routine) terminates according to [10], Algorithm 3 (invoked in line 15) terminates by Lemma 3, and the proof of termination of Algorithm 5 (invoked in line 17) is trivial and follows from the fact that at each iteration, we remove one element from a finite list (*dList*).

We first argue that the while loop from lines 7 to 18 is finite. This while loop terminates if there exists some value *tmp* for some variable i such that (1) *tmp* is valid (line 10), and (2) *tmp* does not give rise to any empty valid domain for other variables neither directly nor during value inference (line 15). The first condition follows from the fact that user chooses *tmp* from a set of values which are valid for i. Consider condition 2, and for the sake of contradiction, suppose assigning any value *tmp* to any variable i results in some empty valid domain. This implies that the reference architecture is unsatisfiable and inconsistent – contradicting our assumption. Hence, the while loop from line 7 to 18 always terminates by assigning some value *tmp* to some variable i.

We then argue that the while loop form line 5 to line 18 terminates. To show that this loop terminates, we argue that the total number of times that the BACKTRACK algorithm is called is finite. This number is less than or equal to the number of all possible value assignments to all the configurable variables. We argued based on Lemma 2 that the total number of configurable parameters is finite. Each configurable parameter is finite domain. Thus, the total number of value assignments to all the configurable parameters is finite. Hence, we call the BACKTRACK algorithm a finite number of times. Therefore, the outer while loop in Algorithm 4 terminates in finite time. □

Theorem 2 (Product consistency and completeness). *Given a consistent reference architecture with no cyclic decomposition relation, Algorithm 4 can always generate a complete and consistent product.*

Proof. Let P be the product returned in line 19 of Algorithm 4. We define completeness and consistency of product P as follows:

- **Completeness:** A product P is complete if all of its configurable parameters are assigned a value.
- **Consistency:** A product P is consistent with its reference architecture, if the six conditions mentioned in Definition 5 hold for it.

The sub-routines CREATENEW and APPLYCONFIGDECISION invoked in lines 1 and 13 of Algorithm 4 guarantee the conformance of product P to the first five conditions in Definition 5. Below, we argue the completeness of P and its consistency with respect to the last condition in Definition 5.

By Lemma 2, the total number of configurable parameters in our work is finite. Let $T = \{p_0, ..., p_m\}$ be the set of all configurable parameters. Due to our configuration assumption, every configurable parameter p_i in T is finite domain. By Theorem 1, Algorithm 4 terminates by assigning some value *tmp* to every variable p_i in T. This proves that Algorithm 4 can always generate a complete product. Further, any value tmp assigned to variable p_i passes the check on line 10 of Algorithm 4. This shows that all the value assignments are consistent, and therefore the result of Algorithm 4 is consistent. □

8 Prototype configuration tool

To empirically evaluate our semi-automated configuration approach, we developed a prototype configuration tool that implements the configuration algorithm given in Section 6.4. Figure 13 shows the architecture of the configuration tool. Inputs to the configuration tool are a model of the reference architecture of the product family, and the user-provided configuration data. The configuration process starts either from scratch by creating an instance of the topmost component type in the reference architecture model, or by loading a partial but consistent configuration file. To collect configuration data from the user, configurable parameters are presented to the user via the *interactive user interface* shown in Figure 14. The configuration tree on the left-hand side of Figure 14 allows the user to freely explore the system-under-configuration and choose the object (i.e., component instance) he wants to configure. Moreover, the configuration engineer can create new components at configuration-time by adding new nodes to the configuration tree. The right-hand side of Figure 14 is used for assigning values to configurable parameters of the selected node of the configuration tree. Depending on the type of the configurable parameter, user inputs can be provided either by entering a value into a textbox (e.g., for configuring cardinalities), selecting an item from a drop-down list (e.g., for configuring the type of a component), or by navigating to a desired node from the configuration tree (e.g., for configuring topology).

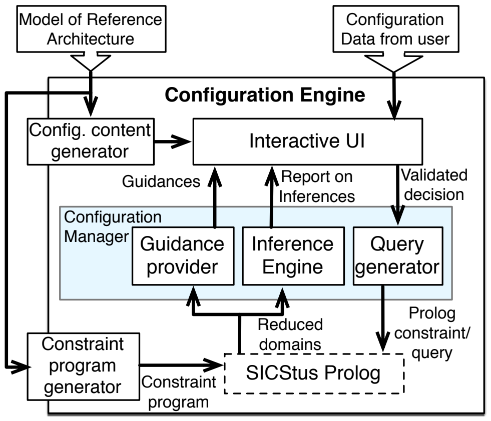
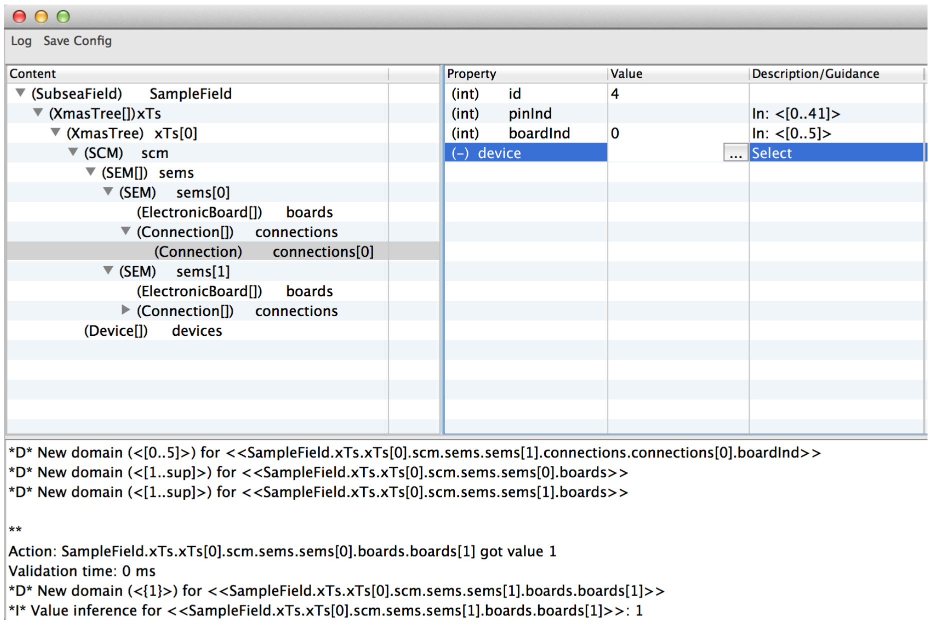


Fig. 13. Architecture of the configuration tool.

For computing the valid domains, we use the clpfd library of the *SICStus Prolog* environment [15, 4]. The finite domains constraint program, in this context, is a Prolog/clpfd program. The clpfd library uses a variant of AC-6, which is an efficient implementation of constraint propagation over finite domains [16]. In our implementation, we use the jasper library that provides an interface for invoking the SICStus Prolog engine from a Java program.

The configuration engine iteratively and interactively collects configuration values from the user. At each iteration, the user provides a value for one configurable parameter. Using the valid domain of the configurable parameter, the con-



```
Content
▼ (SubseaField)      SampleField
  ▼ (XmasTree[])xTs
    ▼ (XmasTree)  xTs[0]
      ▼ (SCM)    scm
        ▼ (SEM[])  sems
          ▼ (SEM)    sems[0]
                (ElectronicBoard[])      boards
            ▼ (Connection[])      connections
                  (Connection)      connections[0]
          ▼ (SEM)    sems[1]
                (ElectronicBoard[])      boards
            ▶ (Connection[])      connections
        (Device[])      devices
```

Property		Value	Description/Guidance
(int)	id	4	
(int)	pinInd		In: <[0..41]>
(int)	boardInd	0	In: <[0..5]>
(-)	device	...	Select

```
*D* New domain (<[0..5]>) for <<SampleField.xTs.xTs[0].scm.sems.sems[1].connections.connections[0].boardInd>>
*D* New domain (<[1..sup]>) for <<SampleField.xTs.xTs[0].scm.sems.sems[0].boards>>
*D* New domain (<[1..sup]>) for <<SampleField.xTs.xTs[0].scm.sems.sems[1].boards>>

**
Action: SampleField.xTs.xTs[0].scm.sems.sems[0].boards.boards[1] got value 1
Validation time: 0 ms
*D* New domain (<{1}>) for <<SampleField.xTs.xTs[0].scm.sems.sems[1].boards.boards[1]>>
*I* Value inference for <<SampleField.xTs.xTs[0].scm.sems.sems[1].boards.boards[1]>>: 1
```

Fig. 14. A snapshot of our prototype configuration tool.

sistency of the configuration value is checked. If the entered value is consistent, the *Query generator* is invoked to update the constraint program accordingly and to start a new constraint propagation session to identify the implications of the user decision on domains of the remaining configurable parameters. The new valid domains serve as input to the *Inference engine*, which implements the inference mechanism explained in Section 6.3. Valid domains are also used by the *Guidance provider* to report to the user the impacts of her decision (e.g., updated valid domains) and to help her make consistent decisions in the subsequent configuration iterations. In the current version of the prototype configuration tool, inferences and the updated valid domains are reported to the user, at the end of each configuration iteration, using the text box in the lower part of the configuration window shown in Figure 14.

9 Evaluation

As discussed in Section 2, one main challenge in developing large-scale ICSs is the issue of incorrect or inconsistent configurations. Identifying configuration errors and debugging configurations are overwhelming tasks, that make development of ICS products costly and time consuming. Configuration errors in large-scale contexts are largely due to the high configuration workload, and the complexity of decision making. Therefore, in our configuration approach, we seek three objectives: (1) ensuring the consistency of full and partial configurations, (2) reducing the amount of manual configuration by automating some configuration decisions, and (3) reducing the complexity of manual configuration by assisting the configuration engineers with the remainder of configuration decisions. In Section 7, we analytically proved that configured products produced by our approach are guaranteed to be consistent. In this section, we empirically evaluate the effectiveness of our approach in addressing the two last objectives. Furthermore, we evaluate the performance of our approach by measuring the response time of the tool. In particular, we answer the following research questions:

RQ1 **What percentage of configuration decisions can be automated using our approach?**
Due to the interdependencies among configuration parameters, a portion of configuration decisions can usually be derived from the previously made decisions. By automating the derivation of these decisions, we can reduce the configuration workload, therefore reducing the likelihood of making inconsistent configuration decisions. Assuming that the workload is proportional to the number of manual configuration decisions, the answer to the first research question indicates how much the configuration workload can be reduced using our approach, therefore providing an insight into how our approach addresses the second objective.

RQ2 **How much do the valid domains shrink at each iteration of configuration (i.e., lines 5 through 11 of Algorithm 4)?**
An answer to this question provides an insight into how much we can reduce the complexity of decision making using the guidance that our approach provides. Recall that in our approach, in each configuration iteration, we present updated valid domains to the user as a form of guidance. The user is always expected to configure a parameter by selecting a value from the associated valid domain. The smaller the valid domains are (i.e., the fewer options the user has), the easier it is to make a decision.

RQ3 **How long does it take, on average, for the configuration tool to complete the computation of one configuration iteration?**
The above question measures the response time of our tool. Since our approach is interactive, its usability greatly depends on the amount of time required to complete individual configuration iteration.

To answer these questions, we designed an experiment in which we used our configuration tool to rebuild three verified configurations from our industry partner that produces subsea oil production systems, as explained in Section 2. All the three configurations belong to a representative product family in the domain of subsea oil production systems. 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. This configuration does not represent a complete product. The other two are the verified configurations of two complete products, which we refer to in this section as Product_1 and Product_2. All the configurations were created from scratch and by following a representative configuration scenario. We logged all the configuration-state changes (i.e., inferences and valid domain recomputations) and execution times in each configuration iteration. A total of 4735 configuration iterations were performed to obtain the results reported in the remainder of this section (i.e., in Sections 9.1 through 9.3).

We performed our experiments using the simplified generic model of the subsea product family given in Section 3.1. The simplified model contains all the main components and consistency rules of the actual product family, but does not go into the details of all parameters of all components. However, it covers most of the defined types of variabilities (the only variability type that is not included is configurable type), and therefore, contains a representative subset of the configurable parameters.

Table 5 summarizes the characteristics of the investigated configurations. Numbers of devices, objects, and variables in Table 5 are calculated w.r.t. the simplified model. Numbers of constraints are extracted from the constraint programs created by the configuration tool. More accurately, the numbers reported in the last column of Table 5 are the numbers of binary constraints created and used by the clpfd library of the SICStus Prolog. Note that the number of constraints in the constraint program changes throughout the configuration process. The numbers reported in Table 5 belong to a random configuration iteration, close to the end of the configuration, with all the required objects being instantiated. Table 6 reports the number of configurable parameters for each distinct type of configurable parameters. These parameters are instances of the configurable features in the simplified model of the reference architecture, where we have one configurable attribute cardinality, two configurable decomposition cardinalities, four configurable attributes, and one configurable topology. As shown in Table 6, a considerably high portion of the configurable parameters are configurable attributes and configurable topologies.

Table 5. Characteristics of the rebuilt configurations.

	# **XmasTrees**	# **SEMs**	# **Devices**	# **Objects**	# **Variables**	# **Constraints**
ESS Test	1	1	111	226	343	223
Product_1	9	18	453	1396	2830	9967
Product_2	14	28	854	2619	5307	37606

Note that the numbers reported in Table 5 and Table 6 would be an order of magnitude larger if we had used a complete model. With the simplified model, it took about half a person-week to rebuild the configurations of the three examples. However, creating actual configurations takes several months and is, normally, performed by a team of expert engineers.

Table 6. Distribution of the types of configurable parameters.

	# **AttributeCard.**	# **DecompositionCard.**	# **Attribute**	# **Topology**
ESS Test	1	5	226	111
Product_1	18	37	1869	906
Product_2	28	57	3514	1708

In Sections 9.1-9.3 we report the evaluation and analysis that were performed on the experiments to answer the research questions. At the end of this section, we further discuss threats to the validity of our results.

9.1 Inference percentage

To answer the first question and identify the effectiveness of our approach in reducing the configuration workload, we have defined an *inference rate* which is equal to the number of inferred decisions (i.e., decisions automatically made by the configuration tool) divided by the total number of configuration decisions:

$$inference\ rate = \frac{inferences}{manual\_decisions + inferences} \tag{3}$$

Table 7 shows the inference rates for each rebuilt configuration.

Table 7. 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. 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.

This experiment shows that our approach can automatically infer a large number of consistent configuration decisions, especially for products with some degree of structural symmetry. Assuming that configuration workload is proportional to the number of manual assignments, our approach can reduce the configuration workload by about 50 % in the case of Product_1 and Product_2. Note that this reduction of workload is calculated with respect to cases where no support for reuse of configuration data is provided. Yet, in practice, primitive support for reuse is usually provided, through the copy-and-paste mechanism.

However, without a gain model specifying the impact of copy-and-paste we cannot compare our approach with the cases where the copy-and-paste mechanism is used to reduce the configuration workload.

9.2 Reduction of valid domains

In large systems, engineers often fail to predict the impact of configuration assignments on other parameters. The larger the systems and the more interdependent the parameters, the more difficult it is for the configuration engineers to predict the implications of their decisions. Providing engineers with parameters' valid domains after each configuration assignment helps them avoid assigning inconsistent values to interdependent parameters, and hence, reduces the chance of creating inconsistent configurations. Recall from Section 6 that the valid domains are pruned after each value-assignment. By doing so, we force the user to choose a value among the remaining valid options. Having to choose from fewer options eases decision-making.

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. Such a comparison is possible and meaningful because all the domains are finite. Table 8 shows the average reduction of domains for each rebuilt configuration. *Reduction rate* in the table is defined as the proportion of the *reduction size* (i.e., number of distinct values removed from a domain during constraint propagations) to the initial size of the domain (i.e., number of distinct values in a domain before constraint propagation). Note that in certain cases, recalculation and reduction of the valid domains may result in value inferences (i.e., when the valid domain is reduced to only one valid value). In the calculations reported in Table 8, we have not considered domain reductions that had resulted in value inferences. Results reported in Table 8 show that the domains of variables can be considerably reduced when a value is assigned to a dependent variable. Specifically, after each value-assignment step, on average, 37.98% of the values of the dependent variables are invalidated. Without such a dynamic reduction of the valid domains, there would be a higher risk for the user to make inconsistent configuration decisions.

Comparing the reduction rates of the three examples reported in Table 8 shows that, as opposed to inference rate that is highly affected by structural symmetry, reduction rate is independent from the structural symmetry of the configured artifacts. More specifically, reduction rates for Product_1 and Product_2, which have structural symmetry, is very close to that of ESS Test, which does not have structural symmetry.

9.3 Response time analysis

The configuration solution presented in this paper is highly interactive. In each

Table 8. 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.				

iteration, the user's decision is validated and its implications are identified and reported to the user. Providing automation as part of such an interactive configuration process requires the underlying computation to be sufficiently efficient for our approach to be practical.

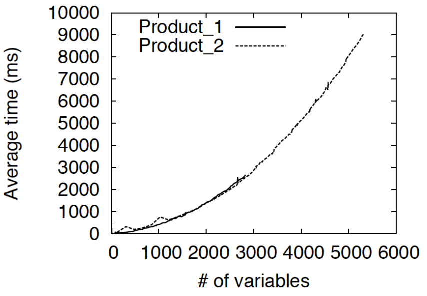


Fig. 15. Constraint propagation time grows quadratically with the number of variables (with a coefficient of determination of 0.9994).

We define the efficiency of our approach as the amount of time needed for validating and propagating the user decision in each iteration. For this purpose, we have measured at each configuration iteration the execution time (i.e., response time), and the number of variables in the constraint program. Figure 15 shows the average time required for executing one configuration iteration. As shown in this figure, for products with less than 1000 variables (each representing one configurable parameter), it takes, on average, less than half a second to validate and propagate the decision. However, this time grows quadratically with the number of variables. It is difficult to compare this result with similar approaches, because to the best of our knowledge the few similar interactive configuration techniques that exist in the literature (see Section 11) do not report on the relation between the response time and the size of the system under configuration (e.g., the number of variables).

Response time measurements reported in Figure 15 suggest that with the current implementation, we can support the configuration of systems with up to a few thousand parameters. However, for some of the real-world applications, where the number of variables can be in the tens of thousands, this quadratic growth of the response time can make our approach inefficient and impractical. In particular, we cannot expect the user to wait several minutes for the configuration tool to validate and propagate a configuration decision. This inefficiency is largely a drawback of our current implementation of the algorithms presented in the paper. In particular, in the current implementation, we are not benefiting from the on-the-fly capability of the constraint propagation technique. This is because the Java interface that we use from SICStus Prolog does not support this capability, although it is implemented in the SICStus Prolog engine. Since the main computation in each configuration iteration is the recalculation of valid domains through constraint propagation, inefficient invocation of the constraint propagation technique drastically impacts the efficiency of our tool. In the current implementation, after the value-assignment in each iteration, all constraints in the constraint program are evaluated, regardless of their relevance to the value-assignment[8], to prune the domains. However, on-the-fly constraint propagation results in the same pruning of the domains, by evaluating only the relevant constraints, which are expected to form a very small portion[9] of the constraints in the constraint program. Therefore, the on-the-fly capability is expected to considerably improve the efficiency of our tool. Improving the implementation of our prototype configuration tool, together with further evaluation and analysis of the efficiency of our tool, are planned to be done in the next step of our work.

10 Discussion

Our configuration approach, which consists of the SimPL modeling methodology [7] and the configuration framework presented in this paper, aims to address the configuration problems that we observed in the domain of integrated control systems in collaboration with an industry partner. While the SimPL methodology targets the ICS domain, the configuration framework is deemed to be more generic, and applicable to all kinds of component-based systems. The current implementation of our configuration framework supports only SimPL models, making the framework applicable to ICSs only. However, our work can be extended (using appropriate transformation steps) to support other component-based variability models, targeting other types of systems. Finally, even though the current implementation is limited to the SimPL methodology and the ICS domain, we don't consider this to be a severe restriction since ICSs already represent a large industry sector.

The SimPL methodology enables the creation of structural and variability models for integrated control systems. As discussed in [7], SimPL models can as well capture behavioral variabilities. However, we note that SimPL models are meant to be used for architecture-level configuration, and are not intended to support requirements- or implementation-level configuration as discussed in [52]. Furthermore, the SimPL methodology does not focus on the needs of other types of software systems, such as data-intensive, networked, or web-based systems.

Since the configuration functionalities in our configuration framework are based on the input reference architecture models, the quality of these models drastically affects the effectiveness of the configuration framework. In particular, using our configuration approach, one can only configure the variabilities that are captured in the generic model of the product family. Similarly, the approach

[8] As mentioned in Section 6.4, such a value-assignment results in adding a constraint of the form $\psi : x = tmp$ to the queue in the constraint propagation algorithm. As shown in line 6 of Algorithm 2, constraints relevant to ψ are the ones that involve variable x.

[9] Calculating the exact percentage of the relevant constraints is not possible using our current implementation.

can validate the decisions and automatically infer decisions only based on the dependencies that are captured in the model. For example, in our earlier work [9], we showed that inference rate increases when we augment SimPL models with additional rules capturing internal similarities. In contrast, we expect that for incomplete models (e.g., models where some dependencies or rules are missing) the quality of validation results and the guidance produced by the configuration framework decreases, and the inference rate to be suboptimal. Therefore, in order to maximize the benefits of using the configuration framework presented in this paper it is crucial to provide high-quality and informative models. Providing such high-quality models although costly, is considered beneficial [7], since such models are created once for a family and then repeatedly used for configuring product instances.

In Section 6.4, we provided an algorithm that uses a basic backtracking mechanism to ensure consistency and completeness of our approach. Backtracking can negatively impact usability and performance of our algorithm. We note that in our industrial case study we were able to configure products in a backtrack free manner. We speculate that we can avoid backtracking during configuration by configuring variables in a particular order. In addition, the backtracking mechanism presented in Section 6.4 can be improved in various ways. In particular, several heuristics in the constraint solving community are proposed that attempt to improve the performance of backtracking. Examples of these heuristics include back jumping and no-good learning [23]. For example, in back jumping, upon reaching an empty valid domain (e.g., line 15 in Algorithm 4), we do not return to the parent in the search tree, but to an earlier ancestor. In no-good learning, corresponding to each sequence of decisions that lead to an empty valid domain, we record a new constraint that characterizes the inconsistent value assignments and use that constraint to detect dead-ends earlier in the future. In future, we plan to characterize the conditions under which backtracking is not used during configuration, and for situations that backtracking is inevitable, we intend to experiment with heuristics such as back jumping or no-good learning to improve performance and usability of our tool. Finally, we note that the logic in Figure 12 does not include some OCL operators (e.g., transitive closure) that we did not require in our configuration problem. In future, we plan to extend our logic and our translation to handle these operators.

11 Related work

Configuration spans a number of domains including artificial intelligence (AI), software product lines (SPL), and formal methods and notations. Below, we compare our work with approaches proposed in each of these domains.

11.1 Configuration in the AI community

Configuration in the AI domain is defined as "*the task of composing a customized system out of generic components*"[49]. This definition matches the notion of configuration defined by the SPL community as well as the one we defined in this paper. Configuration has been a subject of the AI research between 1970 and 1990, where a whole spectrum of configuration problems, mostly related to validation and optimization of configurations, have been studied. Configuration solutions provided by the AI community are composed of a knowledge-base and a reasoner. At an abstract level, knowledge-bases are essentially similar to our reference architecture models, specifying generic components and configuration constraints and preferences. The reasoner is a constraint solver that provides various functionalities (e.g., constraint propagation, consistency checking) to enable consistency checking of the components and their configuration constraints, to identify inconsistencies, and to optimize the consistent configurations for the given preferences.

In contrast to our UML-based reference models, AI knowledge-bases are not described in standard software engineering notations and languages. While we provide a methodology to help domain experts build reference models, there are no guidelines on how to create knowledge bases. In addition, the functionalities provided by the AI configuration reasoners are not tailored to a specific domain, and do not aim to particularly enable configuration use cases, such as interactive guidance, that occur in software engineering applications. In contrast, our work provides an interactive configuration solution aiming at assisting and guiding the configuration engineer throughout the configuration process in the domain of ICSs.

As noted in [45], many of the configuration solutions developed by the AI community have not been taken up by industry. Although a solid analysis of the contributing factors is not available, being general purpose solutions and lacking industrial case reports are deemed to be the main reasons for the industry not to adopt the solutions developed by the AI community. Recent research on configuration (performed by the SPL and the software engineering community) attempts to alleviate this shortcoming by developing domain-specific solutions to improve efficiency and usability [45].

11.2 Configuration in the SPL community

Since 1976, when the idea of software product families was described by Parnas [43], many approaches have been proposed for modeling product families and configuring them. Product families are typically modeled and configured either at the *feature* or requirements level, or at the *architecture* and design level. In this paper, we presented an approach for architecture-level configuration of ICS families. Below, we first compare feature-level and architecture-level configuration approaches. We, then, focus on existing architecture-level configuration solutions to compare them with our work.

11.2.1 Feature-level versus architecture-level configuration

Feature diagrams [35] and their variants [36, 26, 27, 19] have been extensively used for modeling and configuration in product line engineering. These diagrams provide simple means for capturing the variabilities specified at the feature-level. Schobbens et. al. [50, 51, 30, 11] studied the notable variants of feature diagrams and showed that these variants (namely, [36, 26, 27, 19]) are equally expressive. Specifically, all these variants can be represented in terms of a generic abstract syntax proposed in [50, 51, 30, 11]. Feature-level configuration [12], consistency checking of configurations at the feature level [22], and interactive configuration of feature diagrams [5, 33] have been previously studied in the literature for a number of feature diagrams variants.

Feature models, however, are not easily amenable to the architecture-level configuration of ICS families, for which more expressive abstractions (i.e., architecture-level specifications of product families) are needed. In particular, the existing abstractions in feature models fall short in two ways if they were to be used in architecture-level configuration: (1) Feature diagrams support boolean types only, and (2) features and subtrees in a feature tree cannot be instantiated or cloned during configuration. To better illustrate these two shortcomings consider the following: Using the formalisms provided by Schobbens et. al., one can create a constraint program, similar to the one in Section 6, for a feature diagram. Suppose cp be the constraint program created from a sample feature diagram, and suppose cp' be a constraint program created for a sample reference architecture as discussed in Section 6. These constraint programs are different in two ways. First, all the variables in cp are boolean variables, while variables in cp' take their values from finite-domains where variables can take more than two values. Second, the constraint program cp has a fixed size, while the constraint program cp' grows during the configuration. More specifically, as discussed in Section 6, during the configuration of a product in our approach, new variables and new constraints may be added to the constraint program cp' and new values may be added to some of its domains. In the case of feature diagrams, however, the variables and the constraints in the constraint program do not change during the configuration. These differences highlight the challenges of architecture-level configuration compared to feature-level configuration. We have discussed in Sections 6 and 7, how we address these challenges in our semi-automated approach to configuration.

11.2.2 Modeling and configuration at architecture level

In order to capture the complex concepts related to the architecture-level configuration, various extensions to feature models have been proposed, most notably addition of feature cardinalities [18], group cardinalities [48], and feature diagram references and attributes [18]. These extensions are integrated into a more expressive feature modeling notation in [20]. We refer to this integrated notation as the *extended feature modeling* notation, which is known to be as expressive as UML class diagrams [22, 53]. Extended feature models can be combined with OCL constraints or similar constraint modeling languages to form product-family modeling languages that are as expressive as the SimPL methodology.

Extended feature models have been used as a basis for developing a wide range of configuration approaches (e.g., [28, 31, 41, 42, 54, 56]) focusing on various configuration requirements, ranging from validation and consistency checking to configuration scheduling. The closest work to ours are the approaches proposed by Myllärniemi et. al. in [42] and by Mazo et. al. in [41] where architecture-level configuration is enabled via extended feature models, and consistency of the configuration results is ensured using constraint solvers over finite domains. These approaches however differ from ours in the following aspects: (1) they do not allow configuration-time cloning of features – which is identical to the configuration-time creation of components and configurable parameters in our approach (see Section 5.1), and (2) they don't support the verification and the analysis of complex constraints such as those in Section 3.1. Furthermore, the notion of interactive configuration is missing from the approach proposed by Mazo et. al. [41].

The idea of interactive configuration – where implications of user decisions are propagated to avoid incorrect choices – has been previously proposed for original feature models in [5, 33], and for extended feature models in [28, 54, 42]. Similar to our work, all these approaches use constraint satisfaction techniques to enable interactive configuration. However, none of these approaches have addressed the configuration-time creation of new instances (i.e., components, connections, and parameters), its challenges, and its impacts on interactive configuration. In our approach, we have proposed three types of cardinality features to enable configuration-time creation of components, connections between components, and parameters within the context of existing components (see Sections 5.1, and 6.2). Moreover, we have specified how constraint propagation can be applied to determine and propagate the implications of configuration-time creation of new instances.

In [31], Hubaux presents feature-based configuration, which addresses the issue of multi-view multi-stage configuration. The goal is to enable collaborative configuration. For this purpose, views are defined over feature models to establish insulated spaces in which users can safely configure the part of a feature model assigned to them. The approach of [31] further resolves inconsistencies by computing a set of resolutions when a configuration constraint is violated. In our work, however, constraint violations are prevented by interactively guiding the user during the configuration process.

11.3 Formalizing the notion of configuration

An important contribution of this paper is the formalism that we presented in Section 4. Similar formalisms can be found in the literature. In particular, extended feature models and their configuration are formalized in a number of ways (e.g, using grammars [21], and higher-order logic [34]). None of these, however, discuss consistency of the configuration results. Another formalism is presented in [31], where a rigorous formal semantic for extended feature models and the notion of consistency is provided. This formalization focuses on the notion of multi-view staged configuration and the feature-based configuration workflow. In our approach, however, we emphasize on the interactive and iterative nature of architecture-level configuration, and further, describe how the configuration state changes as a result of configuring each type of configurable parameter.

Another related line of work addresses the formalization of UML and OCL (e.g., [47, 38]). In particular, Richters [47] provides an extensive and exhaustive formal semantic for the most-commonly used (if not all) UML constructs. The formalization given in [47] focuses on validating UML/OCL models. Although this formalization is precise and comprehensive, it lacks explicit variability modeling semantics for defining configurable elements. Note that the SimPL methodology extends UML by adding to it essential concepts for variability modeling. Therefore, we chose to base our definitions and proofs on a new formalization (presented in Section 4) that is more amenable to defining configuration logic and configuration consistency.

12 Conclusion and future work

In this paper, we proposed a model-based and semi-automated approach for architecture-level configuration of product-families in the integrated control systems (ICS) domain. Our configuration solution uses a model of the product family's reference architecture and constraint satisfaction techniques over finite domains to provide automation support. The automation support has two major properties: (1) it ensures the consistency of configured products with respect to the input model of the product family's reference architecture, and (2) it reduces configuration effort and complexity of decision making.

We used analytical techniques to prove the first property. In particular, we have presented and formalized the notions of reference architecture, architecture-level configuration, and consistency in our context. A linear-time mapping for casting a configuration problem to a constraint program is provided to precisely describe how constraint propagation over finite domains can be used to ensure the consistency of the configured products. Moreover, we proved that, given a consistent product-line model, our iterative configuration algorithm terminates and can always generate a complete and consistent product.

To show the second property of our configuration solution, we implemented a prototype configuration tool and used it to empirically evaluate our approach. We designed an experiment where we rebuilt three verified configurations of a family of subsea oil production systems to evaluate three important practical factors: (1) reducing configuration effort by inferring configuration decisions, (2) reducing possibility of human errors by reducing the complexity of decision making, 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 correctness and effectiveness of our approach, the value of our tool is likely to depend on its scalability to very large and complex product families. In particular, being an interactive tool, its usability and adoption will very much depend on how fast it can validate configuration decisions and determine their implications in each iteration. Our analysis shows that in our current implementation the propagation time grows polynomially with the size of the product. In future, we will improve our implementation to reduce the propagation time and to improve the scalability of our configuration tool.

The configuration approach presented in this paper does not allow any inconsistencies. In general, however, having some tolerance against conflicts and inconsistencies would be useful. Providing scalable solutions that can be applicable in our context (where normally tens of thousands of variables are configured) is another direction for future work. In addition, we will perform more experiments by applying our approach to build more configurations, and we will do this for various product families developed at our industry partner.

Acknowledgements

The first author acknowledges the Research Council of Norway (the ModelFusion Project - NFR 205606). The second and the third authors are funded by the National Research Fund - Luxembourg (FNR/P10/03 - Verification and Validation Laboratory).

References

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. D. Batory. Feature models, grammars, and propositional formulas. In *SPLC*, 2005.
6. R. Behjati, S. Nejati, T. Yue, A. Gotlieb, and L. Briand. Model-based automated and guided configuration of embedded software systems. In *ECMFA*, 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 the special issue of Inf. Softw. Technol. on Software Product Line Engineering*, 2011.
8. 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.
9. R. Behjati, T. Yue, and L. C. Briand. A modeling approach to support the similarity-based reuse of configuration data. In *MoDELS*, 2012.
10. C. Bessiere. Constraint propagation. Technical report, In, 2006.
11. Y. Bontemps, P. Heymans, P. Y. Schobbens, and J. C. Trigaux. Semantics of foda feature diagrams. In *Workshop on Software Variability Management for Product Derivation – Towards Tool Support*, 2004.
12. P. Borba, L. Teixeira, and R. Gheyi. A theory of software product line refinement. In *ICTAC*, 2010.
13. J. Cabot, R. Clarisó, and D. Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *ASE*, 2007.

14. J. Cabot, R. Clarisó, and D. Riera. Verification of UML/OCL class diagrams using constraint programming. In *ICSTW*, 2008.
15. M. Carlsson and P. Mildner. SICStus Prolog – the first 25 years. *CoRR*, abs/1011.5640, 2010.
16. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *PLILP*, 1997.
17. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* McGraw-Hill Science/Engineering/Math, 2001.
18. K. Czarnecki, T. Bednasch, P. Unger, and U. W. Eisenecker. Generative programming for embedded software: An industrial experience report. In *GPCE*, 2002.
19. K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications.* ACM Press/Addison-Wesley Publishing Co., 2000.
20. K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration using feature models. In *SPLC*, 2004.
21. K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process: Improvement and Practice*, 2005.
22. K. Czarnecki and P. Kim. Cardinality-Based Feature Modeling and Constraints: A Progress Report. In *Workshop on Software Factories at OOPSLA*, 2005.
23. Rina Dechter and Daniel Frost. Backjump-based backtracking for constraint satisfaction problems. *Artif. Intell.*, 136(2), 2002.
24. S. Deelstra, M. Sinnema, and J. Bosch. Product derivation in software product families: a case study. *J. Syst. Softw.*, 74, January 2005.
25. A. Egyed. Instant consistency checking for the UML. In *ICSE*, 2006.
26. M. L. Griss, J. Favaro, and M. Alessandro. Integrating feature modeling with the rseb. In *ICSR*, 1998.
27. J. Van Gurp, J. Bosch, and M. Svahnberg. On the notion of variability in software product lines. In *WICSA*, 2001.
28. T. Hadzic, S. Subbarayan, R. M. Jensen, H. R. Andersen, J. Møller, and H. Hulgaard. Fast backtrack-free product configuration using a precompiled solution space representation. In *PETO*, 2004.
29. 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.
30. P. Heymans, P. Y. Schobbens, J. C. Trigaux, Y. Bontemps, R. Matulevicius, and A. Classen. Evaluating formal properties of feature diagram languages. *IET Software*, 2(3), 2008.
31. A. Hubaux. *Feature-based Configuration: Collaborative, Dependable, and Controlled.* PhD thesis, University of Namur, Belgium, 2012.
32. H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting OCL. *Sci. Comput. Program.*, 44(1), July 2002.
33. M. Janota, G. Botterweck, R. Grigore, and J. Marques-Silva. How to complete an interactive configuration process? In *SOFSEM*, 2010.
34. M. Janota and J. Kiniry. Reasoning about feature models in higher-order logic. In *SPLC*, 2007.
35. 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.
36. 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.

37. C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2), June 1992.
38. Kevin Lano. *UML 2 Semantics and Applications.* John Wiley & Sons, Inc., New York, NY, USA, 2009.
39. F. J. Linden, K. Schmid, and E. Rommes. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering.* Springer-Verlag New York, Inc., 2007.
40. A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1), 1977.
41. 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.
42. V. Myllärniemi, T. Asikainen, T. Männistö, and T. Soininen. Kumbang configurator–a configuration tool for software product families. In *In IJCAI-05 Workshop on Configuration*, 2005.
43. D. L. Parnas. On the design and development of program families. *IEEE Trans. Softw. Eng.*, 2(1), January 1976.
44. G. Perrouin, J. Klein, N. Guelfi, and J. Jézéquel. Reconciling automation and flexibility in product derivation. In *SPLC*, 2008.
45. Charles J. Petrie. *Automated Configuration Problem Solving.* Springer Publishing Company, Incorporated, 2012.
46. K. Pohl, G Böckle, and F. J. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., 2005.
47. Mark Richters. A precise approach to validating uml models and ocl constraints. Master's thesis, Universität Bremen, 2001.
48. M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow. Extending Feature Diagrams with UML Multiplicities. In *IDPT*, 2002.
49. F. Rossi, P. van Beek, and T. Walsh. *Handbook of Constraint Programming.* Elsevier Science Inc., New York, NY, USA, 2006.
50. P. Y. Schobbens, P. Heymans, and J. C. Trigaux. Feature diagrams: A survey and a formal semantics. In *RE*, 2006.
51. P. Y. Schobbens, P. Heymans, J. C. Trigaux, and Y. Bontemps. Generic semantics of feature diagrams. *Comput. Netw.*, 51(2), February 2007.
52. M. Sinnema, S. Deelstra, J. Nijhuis, and J. Bosch. COVAMOF: a framework for modeling variability in software product families. In *SPLC*, 2004.
53. M. Stephan and 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, August 2008.
54. E. R. van der Meer, A. Wasowski, and H. R. Andersen. Efficient interactive configuration of unbounded modular systems. In *SAC*, 2006.
55. D. M. Weiss and R. Lai. *Software product-line engineering: a family-based software development process.* Addison-Wesley Longman Publishing Co., Inc., 1999.
56. Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*, 2012.
57. Lin Xu and Berthe Y. Choueiry. A comparative study of arc-consistency algorithms. `http://www.cs.ubc.ca/~xulin730/mypaper/LXu-ACvsMAC.pdf`, 2001.