To Connect or Not to Connect: Experiences from Modeling Topological Variability

Thorsten Berger
University of Waterloo
IT University of Copenhagen
tberger@gsd.uwaterloo.ca

Stefan Stănciulescu
IT University of Copenhagen
scas@itu.dk

Ommund Øgård
Autronica Trondheim
ommund.ogaard@autronicafire.no

Øystein Haugen
SINTEF Oslo
oystein.haugen@sintef.no

Bo Larsen
IT University of Copenhagen
brla@itu.dk

Andrzej Wąsowski
IT University of Copenhagen
wasowski@itu.dk

ABSTRACT

Variability management aims at taming variability in large and complex software product lines. To efficiently manage variability, it has to be modeled using formal representations, such as feature or decision models. Such models are efficient in many domains, where variability is about switching on and off features, or using parameters to customize products of the product line. However, variability can be represented in the form of a topology in domains where variability is about connecting components in a certain order, in specific interconnected hierarchies, or in different quantities.

In this experience report, we explore topological variability within a case study of large-scale fire alarm systems. We identify core characteristics of the variability, derive modeling requirements, model the variability using UML2 class diagrams, and discuss the applicability of further variability modeling languages. We show that, although challenging, class diagrams can suffice to represent topological variability in order to generate a configurator tool. In contrast, modeling parallel and recursive structures, cycles, informal constraints, and orthogonal hierarchies were among the main experienced challenges that require further research.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages

Keywords

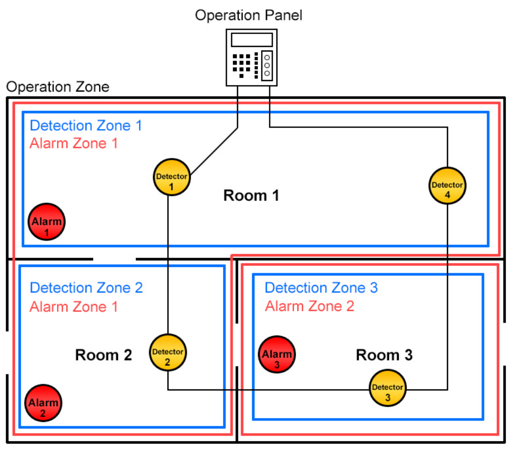
software product lines, variability modeling, topology, configuration, class diagrams, experience report

1. INTRODUCTION

Many companies adopt software product line engineering (SPLE) practices to manage a portfolio of products. Large product lines can have complex variability that needs to be managed. A common technique is to model variability in formal representations—variability models. Popular notations are feature [19, 6, 7] and decision [22, 28] models, which describe the variable and common aspects of products in terms of features or decisions, together with the constraints among them. Variability models are commonly used as input to a configurator tool, which helps users to derive a product by making configuration decisions in an interactive process.

While feature and decision models are effective in a large range of product lines, where variability is concerned with switching features on and off, or with setting parameter values, many domains require richer representations. Consider a fire alarm system, such as the one shown in Fig. 1. It is deployed in a building with three rooms, each representing a so-called fire detection zone. The system is further divided into two alarm zones—the first covering *Room 1* and *Room 2*, and the second covering *Room 3*. Physical devices (alarms and different type of detectors) are connected to a controller or to a fire alarm control panel. When a detection zone is activated by an event, it further notifies the alarm zone that contains that detection zone. Alarm zones can be defined

Figure 1: Example of a fire alarm system



as neighbors, which enables an alarm zone to send alarm notifications to its neighbors.

In our example, we can differentiate between a physical layer and a logical layer. The physical layer is driven by the layout of cables in the building, representing the system structure: different components (detectors, alarms, panels) are wired together in a loop. The logical structure is driven by the building geometry and its use patterns: it is defined as a hierarchy of interconnected detection zones and alarm zones that together specify how alarm detection information is propagated to the users of the building.

Systems with similar topological configuration properties are found in many domains; for example in heating, ventilation, and air conditioning (HVAC) systems, construction of data centers, production lines, and even oil rigs [4]. Complex configurators are built for these systems, such as Novenco Airbox[1], which is a generic configurator for Novenco's HVAC products, or APC's DataCenter Configuration Suite[2], which is used to configure the StruxureWare Data Center's parts.

In general, variability can be modeled with one of many common variability modeling languages, or with a custom domain-specific language (DSL). However, pure feature and decision models, and their corresponding tools (e.g., configurators), are typically not applicable to model topological variability, as they lack concepts to express orders and have multiple instantiations (cloned features or feature cardinalities [8, 9]) of features or decisions. Topological variability cannot be modeled with a fixed number of features, but rather requires entities that can be instantiated and built up inductively during the configuration process. Systems like a fire alarm installation are configured by connecting components in a certain way—variability is represented by a graph instead of a tree.

While there have been comparisons of variability modeling languages and tools, including many variants of feature and decision models, and approaches to develop individual DSLs for variability, only few works have investigated topological variability [12, 11, 3, 24, 16, 4]. The understanding of characteristics and challenges of topological variability is low. There is also very little guidance on modeling, while SPLE practices, such as scoping and core asset development, have been extensively studied and reported by industry.

We believe that commonality in the structure of configurations in the systems considered is significant across vertical domains. This commonality warrants development of horizontal domain-independent methods and tools that are reusable for the entire class of systems. Industries that develop these systems should be able to describe complexity of their installations without hiring language design experts to create domain-specific solutions. Our objective in this paper is to advance this cause by providing a detailed description of an industrial topological variability case, which can be used by researchers to develop such domain-independent solutions.

Our subject is Autronica Fire & Security AS[3], a medium-sized company producing fire alarm systems for almost 50 years. We explore the domain, their existing product lines, characterize topological variability, and model variability using UML2 class diagrams and two off-the-shelf variability modeling languages. Based on the modeling experience, we synthesize a list of experiences about important modeling decisions and a collection of obstacles in modeling such systems using existing technologies. We contribute:

- A description of the fire alarm domain model, including the core concepts and structures in the model, and a set of metrics characterizing the model as a whole.
- Discussion of the major decisions taken, the major difficulties experienced, and the main patterns used during modeling the problem with UML2 class diagrams, the Common Variability Language (CVL), and Clafer.

The paper is focused on the Autronica case, and not on the languages used. We do not attempt an empirical comparison of the languages. The languages have been used as a vehicle to explore the case, and not the other way around. We hope that language designers, configurator designers, and system architecture researchers can use the case in order to provide more robust solutions for this class of problems.

We proceed by summarizing related work in Section 2. Section 3 describes the case company and its system. Section 4 shows the domain model created in collaboration between Autronica, SINTEF, and IT University. Section 5 synthesizes our main observed challenges. Section 6 reports our experiences from using CVL and Clafer. Section 7 concludes.

2. STATE OF THE ART

Haugen et al. [16] use UML2 class diagrams to model configurations of an access control system. Such a system controls user access to a set of access zones, which are controlled by various types of physical access points, which encompass different functionality, communicate with each other, and are organized into logical security levels (e.g., for different floors). Class diagrams are used to model configurations, since configuration is a continuation of the design, so it should use the same concepts. A specific system (a configuration) is modeled based on an architecture diagram using generalization (which allows further specialization, as opposed to instantiation), redefinition (allows specializing the type of a class attribute), subsetting (allows to specialize associations), OCL constraints, and roles (act like subsetting, but appear more readable). We consider this work an important conceptual contribution to the understanding of topological variability, unfortunately so far, no generic tooling has emerged from it.

Svendsen et al. [24] develop a product line of train stations using CVL. Stations are modeled using TCL (Train Control Language), a domain-specific language that is also used to generate interlocking source code for programmable logic circuits [23]. The language controls signals, switches, track circuits, and can allocate different routes for trains. To reduce the manual effort of modeling each station separately in TCL, the authors reverse-engineer a product line for a set of similar stations, which are then controlled using a single CVL model. As such, they show how to effectively manage a selected set of structures, but not how to arrive at a holistic model of a topological configuration space.

Fantechi [12] formulates a general definition of topologically configurable systems, based on the case study of a railway interlocking system. The topology is defined by the layout of tracks, points, and signals in a station or a railway yard. Different interlocking systems, which control the routing of trains, are deployed for different stations according to the defined topology. It appears though that his models are concerned with verification of given station topologies of

[1]http://airbox.novencogroup.com
[2]http://www.apc.com/products/family/index.cfm?id=444&tab=models#specs
[3]http://www.autronicafire.com

finite size (and connections in them), not with modeling an infinite family of possible topologies, and not with supporting engineers in deriving correct instances.

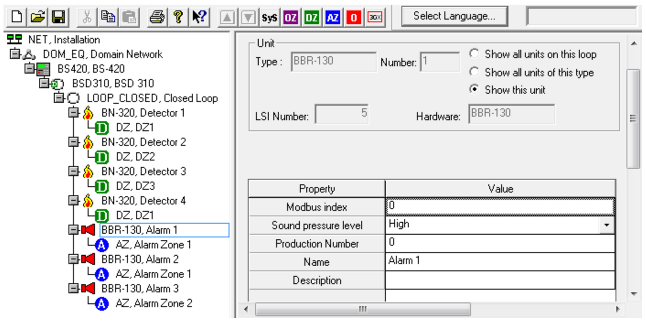
Newer works [4, 3, 11] are perhaps closest to ours considering the objectives. Dhungana et al. [11] discuss topological variability in an industrial logistics system for steel plants. A stockyard contains different types of operational devices, such as cranes, transportation vehicles, and physical spaces for loading and unloading goods, and other areas for different purposes. A hierarchy of stockyards can be created. From a modeling perspective, multiple instantiation and creating (containment) hierarchies is required. Hence, the authors extend their modeling and configuration tool DOPLER [10] to support modeling components with their variability, which is classified into configurable attribute, configurable cardinality, configurable type, and configurable topology. Interestingly, cyclic dependencies are not allowed in the topology (a restriction on instances similar to the one seen in [25]). In contrast, cycles are necessary in our case to realize redundancy as an important safety criteria for fire alarm systems.

Behjati et al. [4, 3] investigate variability in families of integrated control systems (ICS) in the subsea oil production domain. ICS are large-scale systems containing many mechanical, hardware, and software parts. Variability in the considered domain is characterized by instantiation of different types of devices (sensors and valves) organized in a logical (containment) hierarchy, and connected in a topology. Furthermore, tens of thousands of parameters exist, and hardware variability affects software variability, with constraints that cross-cut both spaces. The codebase of an ICS in the domain is typically configured manually in an error-prone process. Further identified challenges in the configuration process are, for instance, configuration guidance as well as the reuse, debugging, and evolution of configurations. To address these challenges, the authors model the domain using a UML profile and propose semi-automatic configuration algorithms, which is a welcomed development in this space. However, their algorithms are limited in the sense that sometimes, these arrive at an inconsistent state and the process needs to backtrack, restarting the configuration at an earlier choice. Also, multiple inheritance is not supported in their method, while it has played a crucial role in our case.

Völter et al. [26] discuss variability in similar domains—water fountains and alarm systems. They argue for DSLs instead of feature models, as the latter are not sufficient to express variability in these domains, which require references and multiple instantiation of features. Furthermore, the models are supposed to express algorithmic logic to some extent, so the ability to write expressions is necessary. Interestingly, the alarm system case study is similar to our domain; unfortunately, no further details and modeling challenges are given; it is not clear whether a topology has to be expressed.

In summary, while solutions for modeling and configuring topological variability are available, there is no in-depth analysis of the variability in the domain of embedded, safety-critical systems. We can see that the ability to model types (to realize multiple instances), hierarchies, and constraints is often reported, but so far the research community has not arrived at robust domain-independent modeling languages and rich off-the shelf interactive configurators for structures.

On a final note, related configuration problems were considered in the area of knowledge-based configuration (KBC). For instance, Felfernig et al. [13] developed an approach to configure UML component diagrams. It supports connecting components in a certain order and types with inheritance; however, cycles and multiple inheritance are not allowed, and the applicability for class diagrams is not evaluated. With recent work [17] that attempts to explore synergies between SPLE and KBC, we see our work as a contribution of a substantial case and requirements to the field of KBC.



```
NET, Installation
  DOM_EQ, Domain Network
    BS420, BS-420
      BSD310, BSD 310
        LOOP_CLOSED, Closed Loop
          BN-320, Detector 1
            DZ, DZ1
          BN-320, Detector 2
            DZ, DZ2
          BN-320, Detector 3
            DZ, DZ3
          BN-320, Detector 4
            DZ, DZ1
          BBR-130, Alarm 1
            AZ, Alarm Zone 1
          BBR-130, Alarm 2
            AZ, Alarm Zone 1
          BBR-130, Alarm 3
            AZ, Alarm Zone 2
```

Type : BBR-130 Number: 1

Show all units on this loop
Show all units of this type
[✓ Show this unit]

LSI Number: 5 Hardware: BBR-130

Property	Value
Modbus index	0
Sound pressure level	High
Production Number	0
Name	Alarm 1
Description	

Figure 2: AutroConf configurator

3. CASE STUDY

Autronica Fire & Security AS is an international manufacturer of safety and emergency management systems, including fire and gas detection, fire suppression, and emergency light systems. Autronica makes and deploys fire alarm systems in a wide range of markets—from small public buildings such as kindergartens, apartment buildings, and others to large-scale industrial installations in the onshore, maritime, petrochemical, oil and gas industry, in buildings and in ships. Manufactured components comprise fire detectors, I/O units, fire panels, and supervisory monitors.

3.1 Subject

Autronica has three main product lines. AutroSafe targets large-scale fire-alarm installations with up to 15,000 units. AutroSafe further comprises advanced features, such as self verification and a dual-path transmission network (AutroNet) for increased reliability. AutroPrime aims at smaller installations in buildings; it has a capacity of 250 units. AutroMaster is a top-level graphical representation system running on one or more PCs. In this paper, we primarily report our experiences from modeling AutroSafe's variability.

3.2 Use Cases and Perspective

Autronica strives for checking rules, regulations, and system constraints at an early stage of the engineering process, well before the delivery starts for each new installation. In the case of fire alarm systems, the configurator not only warrants obtaining the right functionality, but is responsible for enforcing rules required by functional safety certification. Therefore, designing a new AutroSafe installation always involves creating its model. Field equipment is configured by setting various parameters in production and during startup of a panel. In the following, we discuss opportunities and challenges of standardized domain modeling in Autronica.

Modeling configurations using a custom modeling tool. Today, Autronica is handling the configuration data systematically and through proprietary configuration tools. The installation configuration model is built by consultants using a custom configurator tool, which was developed around 15 years ago (see screenshot in Fig. 2). The tool relies on a meta-model expressed in the Entity-Relationship (E/R) notation. The model has evolved over its lifetime, mainly through additions of new physical devices and relationships. The configurator is used to create one central configuration of the complete installation, which is used to generate C-like data structures for each (display and operation) panel.

Unfortunately, the AutroSafe configuration tool is difficult to maintain, partly because it has been tailor-made and does not rely on any modeling or configuration frameworks. Thus, evolving the tool is a burden. It has served well for years, but the infrastructure provides little overview, and requires complex input. UML modeling tools are much easier to use; they are standardized and maintained. The output from these tools can drive more applications than just configuration, and it is accepted by many other tools thanks to standardization.

Capturing topological properties in domain models. In the legacy E/R model, domain properties were described in a very tight way with a high degree of coupling. Hopefully, using a more developed domain modeling language will enable a clear separation between the logical and physical topologies, yet still allow describing the constraints relating the two.

Maintaining configurators and meta-models for similar product families. Presently, configurators for several products exist, but they are independently built and rely on different technologies. Some of the input files use XML, others have a C-like syntax. Even though, the overall configuration procedures are similar for the products families, Autronica does not handle them in a uniform manner.

Since different products have different, yet similar domain models, Autronica needs an effective way to manage a family of related meta-models, and a unified way of deriving topological configurators from them. This requires both a mechanism to derive different meta-model variants from the same base, and/or a suitable structuring approach for the domain model so that commonality and variability between concepts in different product families are organized in a manageable manner. Autronica hopes that standardized domain modeling can decrease cost and increase maintainability of their domain models by managing both in the same process.

AutroSafe and AutroPrime are maintained and further developed in parallel, without a consolidated codebase. Yet, they share a significant amount of concepts common to both; once this commonality is consolidated to unify the configuration, it should eventually lead to merging the codebases.

Exploiting domain models for other applications than configuration. A uniform way of domain modeling for different products will help reusing the same automated test infrastructure across families. Autronica currently develops a simulator, AutroSim, for field equipment that will also have scripting facilities. This opens up possibilities for more automatic testing, which will reduce test time and also enables more extensive regression testing. However, the strong dependency between test procedures and configuration makes it very tedious to write and maintain automatic test scripts. A change in the configuration will also easily lead to a change in both the activation and assessment part of a test script. Autronica's goal is to create a system that decouples test procedures from the actual test rig and configuration.

4. DOMAIN ANALYSIS & MODELING

We performed a domain analysis using action research and focus groups. We collaborated with domain experts and

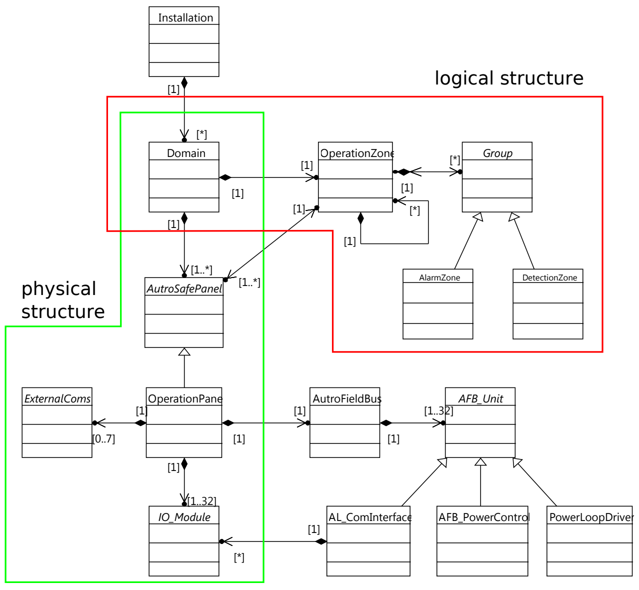


Figure 3: Core parts of physical and logical structures in AutroSafe's meta-model

developers, and also investigated the existing configurator to understand the domain and the variability in the product lines of Autronica. The modeling was done collaboratively in a mixture of local and online meetings.

The primary notation we used was UML2 class diagrams [21] modeled in Eclipse Papyrus. To investigate the feasibility of using variability modeling languages, we also model core parts of the domain in the two languages Clafer and CVL (we report more on these efforts in Section 6). The modeling started in October 2012 and can be divided into four phases:

1. Capture the big picture and the topology of loop units. Followed by a review leading to smaller revisions.
2. Supplement the model with more detailed information, such as attributes and classes to represent concrete devices not used in phase 1. The model was also separated into three packages AutroSafe, AutroPrime, and AutroCommon; the latter by extracting the common concepts of both systems.
3. Verify the model against requirements of processing frameworks, to make it executable for generating an EMF/GEF-based configurator tool.
4. Creation of a new XML format for AutroSafe derived from the domain model (this led to minor changes and adaptations).

4.1 Domain Overview for AutroSafe

A typical AutroSafe installation contains 5–15 panels (the number of panels is limited to 64, most of these are information panels). Detectors and other units are connected in communication loops, of up to 127 (typically about 50). The largest installations in practice reach 10,000 detectors, distributed over one or more detection zones. In extreme cases there are thousands of detection zones, for example one per room or a cabin on a ship. A loop does not have to be contained in one zone (individual devices can be configured to belong to different zones).

The topology is also split into larger alarm zones—areas where alarms are communicated. There are usually much fewer alarm zones than detection zones, but may exist up to hundreds in the worst case. The partitioning of the system into alarm zones is not necessarily compatible with the (finer) partitioning in the detection zones.

To illustrate the domain, we show the core part of the developed domain model in Fig. 3. Its main class is Installation, which represents an AutroSafe deployment. It is composed of multiple Domains, which are meant to separate the fire alarm system into parts that should be independent. That is the case when the parts reside in different buildings, or in installations without any central control or monitoring across buildings.

The remainder of the variability in an AutroSafe system can be classified into a logical and a physical structure. The former aims at organizing devices logically into the zones (e.g., operation zone, detection zone, alarm zone), for instance, to control the activation of alarms or combine fire detectors into groups. The latter comprises the physical layout of devices, such as (control and display) panels, fire detectors and alarms, on the loops.

4.2 Logical Structure

Domains are logically divided into OperationZones. An OperationZone can contain one or more OperationZones itself, which makes it possible to divide a zone into sub-zones, allowing an arbitrarily deep hierarchy of zones.

An OperationZone contains a number of Groups, which are various kinds of collections. The most common groups are DetectionZone and AlarmZone, which can be seen in Fig. 4. Detection zones and alarm zones allow to further divide an operation zone into smaller sections, independent of the actual layout of the installation. For instance, in a building, each room (or groups of rooms) can be a detection zone, and the building could be split up in different alarm zones if only parts of the building should be notified in case of an incident.

Physical detection units are connected to a detection zone. When a detection zone is activated by an event (smoke, flame, fire or gas), it notifies the alarm zones it is connected to (cf., Fig. 4), which will trigger the sound alarm. Two alarm zones that are neighbors can be explicitly configured as neighbors. When an alarm zone starts the alarm, it will notify its neighbors to also trigger the alarm, but with a slightly different sound (to help identify the alarm source).

4.3 Physical Structure

An OperationZone is controlled by one or more Panels. A panel can be either a DisplayPanel or an OperationPanel. While the former just displays information about an operation zone, the latter can both display information and control the fire alarm system. An OperationPanel has modules connected to it, which it controls. An OperationPanel has two categories of modules, external and internal. External modules are used to communicate with components that are not made by Autronica. In the model in Fig. 3, they are represented by ExternalComs and IOModule. Internal modules are used to communicate with components that are made by Autronica, and they are connected using a loop, so it requires a LoopDriver module. The reason for using loops is in case a wire breaks somewhere, the loop driver will still receive information from all the nodes in the loop.

In general *redundancy*, like in using a looped wire instead of a simple one, is a typical design practice in these kinds

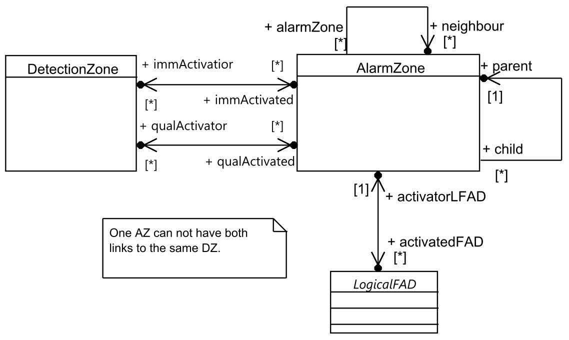


Figure 4: Relationship between detection and alarm zones

of systems—it is enforced by functional safety certification requirements. Redundant components and connections significantly decrease the probability of failure for the entire system (even if individual components are allowed to fail). For the same reason, a LoopDriver has two references to Node, one called primaryNodes and one called secondaryNodes. Thus, for each loop, two loop drivers exist: one primary which only contains primary nodes, and one secondary which only contains secondary nodes. The secondary driver takes over, if the primary one fails.

A Node can then either be a LoopUnit or a Branch. LoopUnits are the physical devices connected to the fire alarm system, such as smoke detectors, alarm sirens, sprinklers, etc. A Node can also be a Branch, which contains loop units itself, but only up to a certain limit since those loop units are not part of the loop so they are more vulnerable if a wire fails.

4.4 Configuration and Deployment Process

AutroSafe is deployed as follows. After the system consisting of detectors, panels, and other components is installed at the customer's location, a static configuration has to be created. This configuration is done by Autronica consultants using the AutroConf configurator (c.f., Fig. 2). One central configuration comprising all operation zones is created. The logical layer is set up by creating zones and establishing relations between them, enabling and configuring physical devices present in the installation The tool then automatically creates a configuration for every panel. After uploading the configurations into the target panels, the system is delivered.

AutroPrime has been developed more recently and shares some of the components and code base with AutroSafe. AutroPrime caters for smaller installations and brings some additional features, such as auto-configuration of the topology. Once the loop units are attached to the system, AutroPrime can provide full functionality at the first initialization. The system auto-configures itself at start-up, creating a configuration based on information from each of the loop units. Its main limitation is that it does not support intricate specifications of logical structures, such as operation zones that can be created in AutroSafe.

4.5 Quantitative Domain Data

We report the main quantitative characteristics of our model, in order to give an overall impression of it, and to explain size requirements for tool builders.

In order to obtain reuse in modeling the two product lines we divided the model into three packages: AutroSafe, AutroPrime and AutroCommon.

AutroSafe. The AutroSafe model consists of 88 classes and over 60 associations between them, including 26 containment relations. The remaining associations are simple references (uni- and bi-directional). The depth of the containment hierarchy reaches up to six levels of classes, including some cycles, so the actual object hierarchies can be deeper.

The model has a very developed generalization hierarchy with 87 relations of this kind (see Section 5.2 for a discussion of this fact). The depth of the generalization hierarchy reaches 3–4 classes, with 11 classes having more than one super-class (multiple inheritance).

In the physical structure, most associations have cardinality constraint of 1 (singleton), but some associations enforce a higher number of possible instantiations (up to 32 or even 100). At the logical layer, most associations do not restrict the number of related objects.

AutroPrime. The AutroPrime model comprises only 69 classes with 54 associations, including 21 containment relations. The containment hierarchy is more shallow, up to 4 classes. AutroSafe generalization is more sparse (64 relations) and its remaining properties are similar to AutroSafe.

AutroCommon. The core part of both systems lies in the AutroCommon package. It consists of 62 classes, and encompasses a total of 9 containment relations and 33 references, plus 60 generalization relations.

The two systems reuse most of the components from the AutroCommon package. The numbers reported for AutroSafe and AutroPrime above have both included the counts of model elements of AutroCommon. As we can see, the amount of reuse is quite substantial. AutroPrime adds very few classes of itself. Furthermore, there is close coupling between the common package and AutroSafe and AutroPrime. Almost half of the AustroSafe system classes have a relation to classes from AutroCommon (either generalization or composition). The same is the case with AutroPrime, exhibiting high coupling between classes of AutroPrime and AutroCommon.

It should be remarked that all above numbers are reported for the model of the core abstract structure of the systems, without classes modeling concrete devices in sale. Adding them would roughly double the size of the models.

4.6 Variability

We can summarize that variability in AutroSafe comprises the following aspects. We distinguish between variability that can be captured by typical feature- and decision modeling languages, and topological variability which requires further modeling concepts.

Feature- and decision-like variability:

- System-wide defaults (e.g., detector and sound defaults, various display options).
- System-wide meta information (e.g., ID number, site name, configuration date).
- Parameters (e.g., texts and identifiers) of concrete physical devices, such as panels. Performance parameters for detectors and detection zones, and timeouts controlling propagation of alarms.
- Sound and pulse patterns for alarm devices, which are saved as 16bit vectors.

Topological variability:

- The connection of LoopUnits (alarms and detectors) to the fire panel.
- The order of loop units on the loop. The panel uses this order to verify that the real and configured topology match;
- The hierarchical organization of operation zones.
- The relationships between alarm zones, detection zones, and operation zones; also the neighbor relationship between alarm zones.
- How sounders and outputs shall be activated in case of an alarm.
- The topology of panels connected via the AutroNet.

5. MAJOR DECISIONS & CHALLENGES

We now discuss the major decision points and major challenges in building our models.

5.1 Relationships

Class diagrams offer three fundamental kinds of relations: containment, generalization, and associations. We used them mostly as prescribed by object-oriented analysis and design, with generalization taking a bit special role of mitigating between logical and physical structures in the model.

Containment. The containment relationships were created to reflect actual physical connections (mostly physical containment). In our model, the contained part does not work if the container is defunct (very close to the UML definition). For example, if you power off the LoopDriver, then its nodes (e.g., LoopUnits such as detectors) stop functioning as well.

However, most modeling tools require a proper containment hierarchy for all objects in the model (not only for objects representing the physical part). Thus, we had to introduce containment relations also in the logical parts, where the choices where much less clear. For instance, AlarmZones hierarchy has not been modeled as a containment hierarchy, but using regular associations (cf. Fig. 4) and all AlarmZones are simply owned by their operation zone (Fig. 3). Such design decisions, caused by technicalities of the modeling language, might be hard to explain to domain experts, who are not language experts.

Associations. We used non-containment associations to model logical dependencies, for example, defining which detection zones trigger alarms in what alarm zones (Fig. 4). Some physical connections could not be mapped to containment, so we used regular associations for them (see Section 5.3).

Generalization is used to classify different kinds of units in the system, both with respect to their physical connectivity and with respect to their logical functions (see below). Generalization diagrams (like Fig. 8) have been used a lot to aggregate information about different kinds of units available.

5.2 Separating Logical & Physical Structures

We quickly realized that it is beneficial to separate logical and physical structures in the model. The challenge was though, that these two structures are defined over the same objects. Since an object can only have one container, it was clear that separation of logical and physical structures also requires separation of the two containment hierarchies—only one of them can actually contain the concrete classes. We decided to place them in the physical structure hierarchy.

In Fig. 5, the classes representing different kinds of devices are placed in the bottom, in a fragment of a generalization hierarchy. Classes representing concepts of physical and logical structures are placed at the very top. Observe now that the class ExternalFAD comprises objects that are contained by

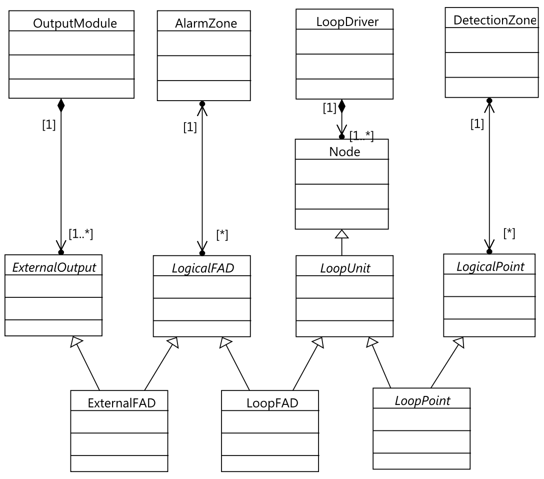


Figure 5: Inheritance hierarchy inflation due to separate logical and physical containment hierarchies

OutputModule, whereas all subclasses of Node are contained by a LoopDriver. Both OutputModule and LoopDriver are concepts from the physical topology. At the same time, there is no containment (no black diamond) from classes representing logical concepts (AlarmZone and DetectionZone).

Furthermore, the same object can enter different kinds of relations in the two structures. This requires a more intricate generalization hierarchy if we want to use types to restrict legal connections: one collection of types decides what relations one can enter in the physical layer, and another does this for the logical structure. These two collections are related tightly using multiple inheritance in a single generalization graph.

In our example, the class LogicalFAD comprises devices that can be connected to alarm zones (sounders and alike), and LogicalPoints comprise devices that can be connected to detection zones (fire detectors and alike). These two classes partition the set of nodes correspondingly into LoopFADs and LoopPoints (Fig. 5). Even though, objects of the two classes can be connected to the same place in the physical topology (to a LoopDriver), they are different with respect to their functions in the logical structure.

Using this modeling pattern multiplies the number of classes quite significantly. If we need to distinguish n types in one layer, and m types in another layer, we might need $n \times m$ types in the worst case. Clearly, this would not scale well if more than 2–3 structures need to be imposed in the same system. The pattern is also quite complex, even though the use case of several structures over the same sets of components is quite common for topological systems.

5.3 Loop Topology

The central part of the physical structure is the loop topology of the main communication network: a cyclic communication wire is connected to a controller (called a driver here). Individual devices can be put on the wire, as well as a certain amount of limited length branches (which are not loops, but spikes). Controllers are duplicated to increase fault tolerance and switched automatically by a watchdog device (AutroKeper). This topology is representative of a fail-safe communication sub-system in safety-critical installations [18].

Refer to Fig. 6 for our modeling. A LoopDriver owns a

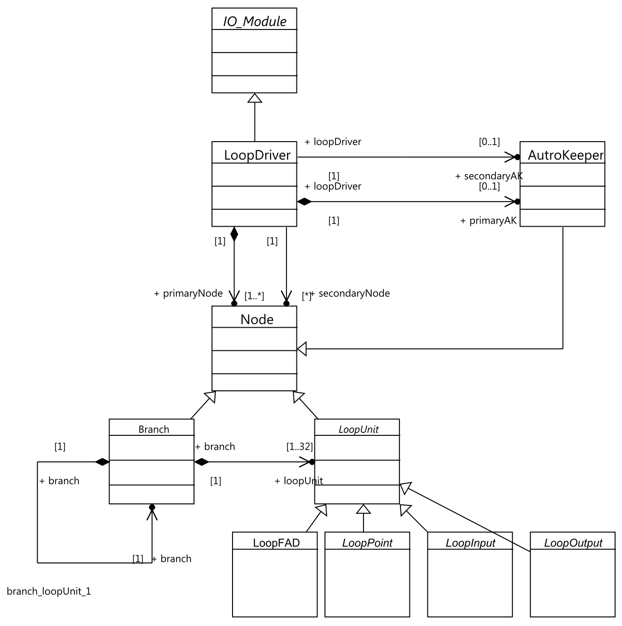


Figure 6: Loop topology

Node, which can be a LoopUnit (a fire detector or an alarm device), a Branch, or a so-called AutroKeeper. The latter, as part of the fail-safe system, allows the loop to be controlled (and powered) by two LoopDrivers—a primary (active) and a secondary (inactive) one. This topology induces complex constraints (discussed further in Section 5.4).

Observe that in our model, the loops are actually not modeled as cycles, but as ordered collections (the primaryNode and secondaryNode properties of the LoopDriver). The order of the components in the loop is important, as the panel checks that the topology of the physical installation and the loaded configuration's topology are the same.

Finally, the nodes are contained in the primary LoopDriver and not in the secondary one (similarly for the AutroKeeper). This is a certain asymmetry, caused by the limitation of class diagrams that each object only has one owner. We observe that parallel structures are quite typical in safety-critical systems to support redundancy, yet there is no specific support for them in the class diagram language, nor in the variability modeling languages known to us.

5.4 Constraints

Most of the domain constraints are represented by cardinalities in the model. The remainder of the constraints are either simple cardinality constraints that cannot be expressed in the structure (e.g., at most eight MimicPanels in AutroPrime, see Fig. 7), simple physical constraints (e.g., maximum use of power or cable length), or more complex constraints. Modeling them is necessary and desired by Autronica to provide proper configurator support. So far, many constraints are hard-coded in the legacy configurator tool.

Our experience with defining constraints was ambivalent. Formulating them in a precise textual form was difficult. Therefore, we did not use any formal notation to capture the constraints. Consider these examples:

- *The Nodes are either owned by one LoopDriver and not referenced by any other LoopDriver, or owned by a pri-*

mary LoopDriver and referenced by a secondary LoopDriver. The primary and secondary LoopDrivers are owned by different OperationPanels. All nodes owned by one primary LoopDriver are also referenced by the same secondary LoopDriver. The Node ordering is different for primary and secondary LoopDrivers.

- *ExternalOutput/Input: Cardinality is 4 if module type is BSB-310 or BSE-310, and 8 for BSJ-310 and BSE-320.*
- *ExternalOutput of type FWRE or CommonTrouble shall be at position 4 if moduleType is BSB-310.*

The first constraint defines the fail-safe functionality of the loop topology. The second and the third are imposed by concrete physical devices. The latter case is very frequent.

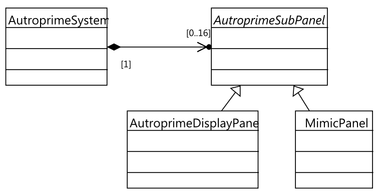
In our experience, formulating constraints during an intensive high-level modeling session is a very disruptive activity. Writing them in a formal language is essentially impossible in such a setting. Thus, any proper modeling tool should facilitate capturing constraints in natural language, as an exact and complete formulation for all constraints is unlikely. It should be later possible to refine them into formal statements outside the main modeling meetings. A formal modeling language that mixes informal and formal specifications for structural and behavioral models is Buiness Object Notation (BON) [27]; it would be interesting to try these kind of languages in modeling experiments.

Further, investigating the expressiveness, conciseness, and reasoning support of appropriate constraint languages (such as OCL) for this domain would be an interesting future work. We conjecture that, given that redundancy is a very common requirement in safety-critical systems, at least quantifiers should be supported.

5.5 Physical Device Library

Recall that the concrete physical devices (around 50) are not modeled in the domain model. In our figures, we stop at the level of their direct super-classes, as shown in Fig. 8. Conceptually, classes representing concrete devices should be part of an external library, which is maintained separately, perhaps even outside the model, in a database. If such separation was successful, it would allow adding new devices to the portfolio without involving modeling experts. This vision might be not realistic, though. While putting configuration parameters for each device into the library is simple, many devices have constraints that cross-cut the model. Autronica has experienced the need to define new constraints in the system, when just one new device was added for a specific market. Moreover, adding new devices can change the logical structure, for instance, the way how inputs and outputs are related could introduce new groups, including different rules for calculating outputs depending on the inputs. It is not

Figure 7: Simple cardinality constraint not expressible in structure: at most eight MimicPanels in AutroPrime



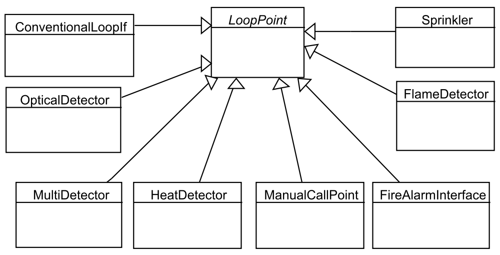


Figure 8: Physical device super-classes

clear then how to handle such changes without further modeling sessions, and whether lightweight portfolio extensions are at all possible.

5.6 Specialized Configurators

Generating domain-specific configurators for AutroSafe and AutroPrime is one prospective benefit of having an integrated domain model. Having specific sub-configurators for parts of the domain model is another challenge. Specifically, a tool to configure the complex (and often large-scale) loop topology would be beneficial. It could rely on a graphical representation, instead of a tree-based visualization used for the logical structures also in the legacy tool. Each of these configurators would have to create a partial configuration, which would need to be integrated into a complete configuration. Research on this topic exists for feature- and decision models [5, 9], but realizing such an approach for topological variability, potentially in combination with a variability model layer that is used to generate dedicated configurators, has not been done so far.

6. OTHER MODELING LANGUAGES

To broaden our experience with modeling the case, we experimented with domain modeling in two further languages, *Clafer* [2] and the *Common Variability Language* (CVL) [1]. We chose them for two reasons. First, they offer concepts required for the topological variability, such as multiple instantiations (feature cardinalities [8]), polymorphism, and references. Second, we had the necessary expertise in the team, as two of the authors were involved in the development of these languages. Both languages support modeling of the problem space, the solution space, and the mapping between both. In Clafer, all three components are defined using the same textual syntax, and usually in one model. CVL offers a feature-models-like problem space language (called VSpec tree) and a mapping language that allows to define variation points over any MOF-based model [20], representing the solution space (called *base model*). We used both languages to model core parts of our domain. In the following, we discuss the main experiences. We refrain from giving Clafer examples, which would require explaining the textual syntax and semantics. For CVL, as a visual language, we provide figures for illustration.

The main difference in modeling topological variability between Clafer and CVL lies in the choice of the deductive or inductive paradigm of describing the instances. Clafer follows the same principle as class diagrams: the entire family of topologies is captured in one model, from which instances can be deductively derived by constraining the properties of the entire family. In such a language, the domain model

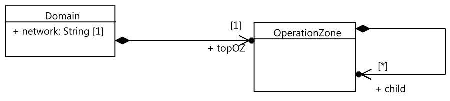


Figure 9: Operation zone hierarchy

represents the variants implicitly, in an aggregate form. CVL belongs to a family of languages that capture the variants explicitly (other examples are [15, 14]). In such languages, the family is based around a central *instance* and a number of variations that inductively derive variants from these instances (so the instances are specified explicitly, while there is no aggregate explicit representation of the solution space). This distinction is interesting, as we find that engineers often find it easy to point out differences between variants explicitly (inductively), however, it seems that modeling this way is often more cumbersome for modeling experts.

6.1 Clafer

As Clafer is a mixture of class and feature models with a unified semantics, modeling most classes and relationships was straightforward with it. Clafer allows to naturally mix-in feature-modeling style variability directly into the class model, which we consider a significant advantage. Clafer can also use syntactic nesting to represent containment, which creates much more concise models than class diagrams.

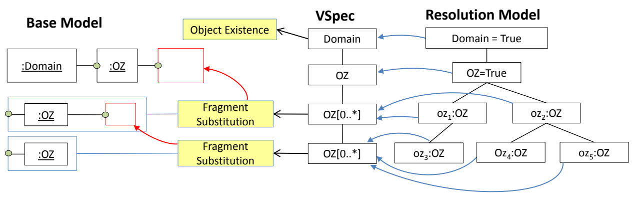
Unfortunately, in our case, the contained classes participate in inheritance hierarchies (see for example LoopUnit in Fig. 6). Presently, Clafer does not allow inheritance from nested classes. This forces flattening of the models and looses one of the major advantages of the language.

Another challenge is that Clafer does not support multiple inheritance, so using the pattern of Section 5.2 to overlay two different structures over the same set of classes is not possible. One needs to resort to more cumbersome patterns, like replacing inheritance with aggregation and proxy objects: instead of having a complex generalization hierarchy that precisely describes the roles of types in the structures, one can represent roles by objects that serve as references to the right types using multiple associations (and then constrain them to only refer to one object at a time). This would lead to a much higher number of references in the model.

6.2 CVL

For CVL, we created a minimal variability abstraction model (problem space) together with a mapping and a minimal base model inspired by our domain model. As indicated above, an important difference with modeling using CVL, as opposed to class diagrams or Clafer, is that the base model is not a representation of a domain, but a representation of a reference instance, so it is an object model capturing an initial installation. This can be seen in CVL examples: in

Figure 10: Operation zone hierarchy in CVL



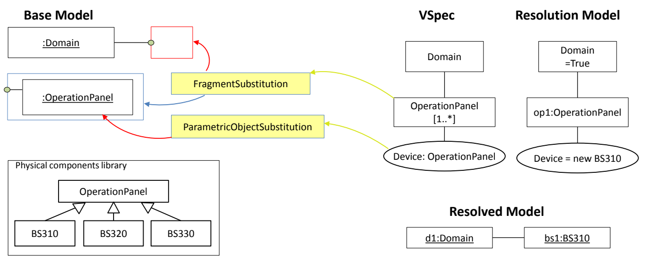


Figure 11: Separated library of physical devices in CVL

Fig. 10 (top-left) and in Fig. 11 (left), the base models are object diagrams, not class diagrams.

Our first observation was that using a minimal installation and growing it with variation points is very difficult for base models that contain a lot of references (like our topologies). It leads to models that are very abstract, programmatic in nature, and lack the simplicity of just indicating differences appreciated by domain experts. This was an interesting experience, as little is known about good base models for inductive variability modeling languages.

Second, modeling arbitrary depth hierarchies was hindered by lack of recursive constructs in the feature modeling part of CVL. Fig. 9 recalls the class diagram of OperationZone hierarchy. While modeling this hierarchy is simple in class models, it is not possible in common feature modeling languages. CVL suffers from the same limitation, since its iteration constructs only allow creation of flat lists. We had to limit the depth of systems to a constant size and explicitly unroll the hierarchy to this depth. Fig. 10 shows an example of this approach. The central part (VSpec) shows a feature model with nesting of operation zones (OZ) up to three levels. Note that there may be arbitrary many operation zones in this model, but they can only be nested in the predefined depths. A resolution representing a particular installation is shown to the right (with five operation zones). In the left part of the figure, fragment substitution variation points are used to instantiate an operation zone object for each configured operation zone in the feature (VSpec) hierarchy. The base model would be able to accommodate arbitrary nesting, but the VSpec hierarchy limits the practically available depth.

In contrast to this experience, CVL has easily supported a separated concrete device library, although the challenge with cross-cutting constraints (cf., Section 5.4) is not solved. Fig. 11 shows the CVL model for replacing abstract objects with concrete physical device objects. In the base model (left), the OperationPanel will be replaced (variation points FragmentSubstitution and ParametricObjectSubstitution) by objects corresponding to the concrete physical devices (BS310, BS320, and BS330).

7. CONCLUSION

We have presented an experience report on modeling topological variability in the safety-critical domain of fire alarm systems. Our aim was to contribute requirements, structures, and challenges in a kind of variability that cannot be modeled using off-the-shelf feature- and decision-modeling languages. We hope that the quantitative data about the model, and the detailed excerpts can provide requirements for tool developers, language designers, and researchers.

Topological variability in our case required the use of types, multiple inheritance, containment hierarchies, associations, ordered collections, and cardinality constraints. These concepts were necessary to model two main structural characteristics of the domain: (i) two orthogonal hierarchies (logical and physical structure) over the same objects (detectors and alarms), and (ii) parallel structures and cycles to represent fail-safe functionality, as common in safety-critical systems.

Our choice of the three very different languages—UML2 class diagrams, Clafer, and CVL—shows that class diagrams were surprisingly well suited to represent the domain and its variability, while our experience with the variability modeling languages Clafer and CVL was mixed.

In summary, we faced the following challenges that are currently not sufficiently addressed by modeling tools and languages. To be effective for topological variability, these should:

- allow more than one containment hierarchy;
- offer natural constructs for expressing redundancy of structures;
- allow working with formal and informal constraints—for instance, to capture constraints informally and then to refine them to a formal notation;
- support recursive structures and multiple inheritance (our CVL and Clafer models suffered from their lack);
- support a database with elements that change most often on the lowest abstraction level;
- enable domain-specific configurators that can use suitable visualizations for different parts of the model and manage partial configurations.

8. ACKNOWLEDGEMENTS

Supported by ARTEMIS JU (grant n° 295397) and the Danish Agency for Science, Technology and Innovation.

9. REFERENCES

[1] Common variability language (CVL). OMG revised submission. 08 2012.

[2] K. Bak, K. Czarnecki, and A. Wasowski. Feature and meta-models in clafer: Mixed, specialized, and coupled. In *SLE*, 2011.

[3] R. Behjati, S. Nejati, and L. C. Briand. Architecture-level configuration of large-scale embedded software systems. *ACM Trans. Softw. Eng. Methodol.*, 23(3):25:1–25:43, June 2014.

[4] R. Behjati, T. Yue, L. Briand, and B. Selic. Simpl: A product-line modeling methodology for families of integrated control systems. *Inf. Softw. Technol.*, 2013.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6), 2010.

[6] T. Berger, R. Rublack, D. Nair, J. M. Atlee, M. Becker, K. Czarnecki, and A. Wasowski. A survey of variability modeling in industrial practice. In *VaMoS*. 2013.

[7] T. Berger, S. She, R. Lotufo, A. Wasowski, and K. Czarnecki. A study of variability models and languages in the systems software domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013.

[8] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1), 2005.

[9] K. Czarnecki and C. Kim. Cardinality-based feature modeling and constraints: A progress report. In *International Workshop on Software Factories*, 2005.

[10] D. Dhungana, P. Grünbacher, and R. Rabiser. The dopler meta-tool for decision-oriented variability modeling: A multiple case study. *ASE*, 18(1), 2011.

[11] D. Dhungana, H. Schreiner, M. Lehofer, M. Vierhauser, R. Rabiser, and P. Grünbacher. Modeling multiplicity and hierarchy in product line architectures: Extending a decision-oriented approach. In *WICSA*, 2014.

[12] A. Fantechi. Topologically configurable systems as product families. In *SPLC*, 2013.

[13] A. Felfernig, G. E. Friedrich, and D. Jannach. Uml as domain specific language for the construction of knowledge-based configuration systems. *International Journal of Software Engineering and Knowledge Engineering*, 10(04):449–469, 2000.

[14] A. Haber, K. Hölldobler, C. Kolassa, M. Look, B. Rumpe, K. Müller, and I. Schaefer. Engineering delta modeling languages. In *SPLC*, 2013.

[15] A. Haber, C. Kolassa, P. Manhart, P. M. S. Nazari, B. Rumpe, and I. Schaefer. First-class variability modeling in matlab/simulink. In *VaMoS*, 2013.

[16] O. Haugen and B. Møller-Pedersen. Configurations by UML. In *EWSA*, 2006.

[17] A. Hubaux, D. Jannach, C. Drescher, L. Murta, T. Männistö, K. Czarnecki, P. Heymans, T. Nguyen, and M. Zanker. Unifying software and product configuration: A research roadmap. 2012.

[18] International Electrotechnical Commission. Functional safety of electrical/electronic/programmable electronic safety related systems. IEC 61508, 2000.

[19] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical report, 1990.

[20] Object Management Group. Meta-object facility. http://www.omg.org/spec/MOF/2.0/.

[21] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[22] K. Schmid, R. Rabiser, and P. Grünbacher. A comparison of decision modeling approaches in product lines. In *VaMoS*, 2011.

[23] A. Svendsen, G. K. Olsen, J. Endresen, T. Moen, E. Carlson, K.-J. Alme, and O. Haugen. The future of train signaling. In *MoDELS*, 2008.

[24] A. Svendsen, X. Zhang, R. Lind-Tviberg, F. Fleurey, Ø. Haugen, B. Møller-Pedersen, and G. K. Olsen. Developing a software product line for train control: A case study of CVL. In *SPLC*, 2010.

[25] E. R. van der Meer, A. Wasowski, and H. R. Andersen. Efficient interactive configuration of unbounded modular systems. In *SAC*, 2006.

[26] M. Völter and E. Visser. Product line engineering using domain-specific languages. In *SPLC*, 2011.

[27] K. Walden and J.-M. Nerson. *Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems*. Prentice-Hall, 1994.

[28] D. M. Weiss. Software synthesis: The FAST process. In *CHEP*, 1995.