Engineering Virtual Domain-Specific Service Platforms

Specific Targeted Research Project: FP7-ICT-2009-5 / 257483

Open Variability Modelling Approach for Service Ecosystems

Abstract

Creating domain-specific service platforms requires the capability of customizing and configuring service platforms according to the specific needs of a domain. In this deliverable we address this demand. We focus on how to describe customization and configuration options in service (platform) ecosystems using a variability modelling language.

The focus is on developing a variability modelling language specific to the INDENICA project. For this purpose, we analyze variability modelling requirements of different categories, ranging from basic to INDENICA-specific, discuss in detail the expressiveness of existing modelling concepts, introduce advanced modelling concepts related to service-based systems, and, finally, describe the concepts of the INDENICA variability modelling language (IVML) to describe customization and configuration options in service (platform) ecosystems.

Document ID:	INDENICA – D2.1
Deliverable Number:	D2.1
Work Package:	WP2
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2011-12-31
Author(s):	SUH, SAP, SIE, PDM, TEL, UV

Project Start Date: October 1st2010, Duration: 36months

Version

1.0	31. January 2012	final version

Document Properties

The spell checking language for this document is set to UK English.

Table of Contents

Table of Contents 3
Table of Figures 5
1 Introduction 6
2 Demands for Variability Modelling in INDENICA 8
2.1 General Variability Modelling Requirements 8
2.2 Variability Modelling in Service-Based Systems 11
2.3 Requirements from Industry 15
2.4 Summary 16
3 Core Variability Modelling Concepts 20
3.1 Running Example 20
3.2 Basic Variability Modelling 21
3.3 Cardinality-Based Variability Modelling 23
3.4 Non-Boolean Variability Modelling 24
3.5 Configuration References 27
3.6 Domain-Specific Languages 29
3.7 Summary 32
4 Advanced Variability Modelling Concepts 34
4.1 Service Ecosystems 34
4.2 Quality of Service and Service Level Agreements 36
4.3 Meta-Variability 38
4.4 Service Technology-Specific Extensions 39
4.5 Summary 41
5 The INDENICA Variability Modelling Approach 42
5.1 INDENICA Variability Modelling Core Language 43
5.1.1 Projects 43
5.1.2 Types 44
5.1.2.1 Basic Types 44
5.1.2.2 Enumerations 45
5.1.2.3 Container Types 45
5.1.2.4 Type Derivation and Restriction 46
5.1.2.5 Compounds 47

5.1.3 Decision Variables 48
5.1.4 Constraints 50
5.1.5 Configurations 52
5.2 Advanced Concepts of the INDENICA Variability Modelling Language 53
5.2.1 Attributes 53
5.2.2 Advanced Compound Modelling 55
5.2.2.1 Extending Compounds 55
5.2.2.2 Referencing Elements 56
5.2.3 Advanced Project Modelling 58
5.2.3.1 Project Versioning 58
5.2.3.2 Project Composition 59
5.2.3.3 Project Interfaces 61
5.2.4 Advanced Configuration 63
5.2.4.1 Partial Configurations 63
5.2.4.2 Freezing Configurations 64
5.2.4.3 Partial Evaluation 66
5.2.5 Including DSLs 68
5.3 Fulfillment of Requirements 69
5.4 Summary 73
6 Application of Concepts in the INDENICA Case Studies 74
6.1 Application of the Core Language 74
6.2 Usage of Advanced Concepts 77
6.3 Overview on IVML Concepts used in INDENICA case studies 81
7 Conclusion 83
References 85

Table of Figures

Figure 1: Running example using basic feature modelling. 22
Figure 2: Running example using cardinality-based feature modelling. 24
Figure 3: Running example with feature attributes and non-Boolean constraints. 25
Figure 4: Replicated sub-trees in the running example (model fragment). 26
Figure 5: Replicated sub-trees using feature cardinalities (fragment). 27
Figure 6: Running example with configuration references. 29
Figure 7: A DSL for configuring content sharing applications (fragment). 31
Figure 8: Example instantiations of the DSL in Figure 7 (fragment). 32
Figure 9: DSL extended by OCL constraints and primitive Java datatypes (fragment). 32
Figure 10: Fragment from the variability model of the Remote Maintenance case study. 74
Figure 11: Simplyfied fragment from the WMS variability model 75
Figure 12: Simplified fragment form the YMS case study. 76
Figure 13: Configuration of decision variables in the WMS case study. 76
Figure 14: Fragment of the compile time variability model from the Remote Maintenance case study. 77
Figure 15: Fragment of the composed runtime variability model from the Remote Maintenance case study. 78
Figure 16: Compound refinements in the WMS variability model. 79
Figure 17: Freezing configuration values in the WMS case study. 80
Figure 18: Combination of partial evaluation and freezing a configuration in the WMS case study. 80

1 Introduction

The main focus of work package 2 within the INDENICA project is the customization of service platforms. As part of this effort, this deliverable addresses variability modelling, variability modelling in service-based systems in general and open variability modelling approaches for service ecosystems in INDENICA in particular. This is of course related to the implementation of variability, which we addressed in deliverable D2.2.1 initially, and which we will finalize in deliverable D2.2.2.

In this deliverable the modelling of variability in general as well as variability modelling in service-based systems serves as a basis for determining the core expressiveness of the INDENICA variability modelling language. This core variability modelling language will be further developed and enhanced by extensions that arise from the specific requirements of the INDENICA project. As a consequence, the structure of this deliverable follows the distinction of core modelling aspects and advanced variability modelling extensions.

In Section 2, we define the requirements for variability modelling in INDENICA. This information is derived from multiple sources, including general variability modelling requirements, demands for variability modelling in service-based systems and feedback from the industrial partners in the project.

Section 3 discusses core variability modelling concepts. The focus of this section is to analyse modelling concepts of different expressiveness with respect to their support for service-based systems as required in the INDENICA project. The results of this discussion will serve as a basis for the core of the INDENICA modelling language that we will define in Section 5. The discussion will cover basic variability modelling using Boolean elements and expressions, the introduction of cardinalities, the extension to non-Boolean expressions, configuration references, and finally the integration of domain-specific languages (DSLs). We will approach this discussion by investigating the expressiveness of each concept. The investigation focuses on the modelling elements introduced by the specific concepts and the different constraint and operator types available for dependency management. A running example will illustrate the various levels of expressiveness in variability modelling throughout this section.

In Section 4, we discuss advanced variability modelling concepts. The focus of this section is to provide an overview of additional extensions to core variability modelling that are required by the INDENICA project. The results of this discussion will serve as a basis for the advanced variability modelling concepts of INDENICA that we will define in Section 5. These extensions will satisfy the additional requirements that the INDENICA core variability language does not directly address. We will cover issues like support for service ecosystems, Quality of Service (QoS) and Service Level Agreements (SLA), meta-variability, and service technology-specific extensions.

Section 5 defines the INDENICA variability modelling approach. The investigation of variability modelling concepts with different expressiveness in Section 3 serves as a basis for the first part of this section. We describe a core variability modelling

language that relies on our analysis of the required expressiveness. The second part then describes the auxiliary features of the variability modelling language. This in turn relies on the analysis given in Section 4.

Finally, Section 6 reviews the concepts of the INDENICA variability modelling approach introduced in Section 5 from the point of view of the INDENICA case studies. We will discuss the usage of individual language concepts in this section as well as the actual coverage of the approach by the (preliminary) variability models that were developed as part of the work on the use cases so far.

Further relationships to other INDENICA deliverables are:

- D1.2.1: Variabilities in the requirements model.
- D3.1: Variabilities in architectural models and in the view-based modelling approach, instantiation of models and variable assets by generative techniques
- D4.1: Configuration of deployment and monitoring.
- D5.2: Concrete variability points in the use cases and industrial platforms.

Comments on the relation to previous work:

- All analysis performed in Sections 2-4 have been performed exclusively as part of the INDENICA project and were motivated by the project.
- The specific proposal for a variability-modelling approach developed in Section 5 was derived specifically within and for the INDENICA project. Relationships to previously developed variability modelling approaches exist. They are further discussed in Section 5.

2 Demands for Variability Modelling in INDENICA

In this section, we provide an overview of the main requirements that we could identify for variability modelling in the INDENICA context. These requirements were derived from several sources. Some requirements were already described on a high level in the proposal document. Some requirements are implicitly stated in the description of the INDENICA case studies in Deliverable D5.1. We will cross-reference the case study requirements here in terms of footnotes. We will first consider variability modelling in general as a starting point to derive basic demands for such modelling approaches. The results of this consideration and analysis are described in Section 2.1.

In Section 2.2, we will investigate the demands for variability modelling that arise in the context of service-based systems in general, and, in particular, in the context of the INDENICA project. This will also satisfy the high level requirements of the proposal document.

In addition, we discussed variability modelling requirements with our industrial partners. While these requirements are somewhat influenced by existing modelling approaches and experiences in variability modelling, they should provide additional insight. This will be described in Section 2.3.

2.1 General Variability Modelling Requirements

A variability model is an abstraction of all common and varying software assets, for example, of a specific software product line. The model illustrates all commonalities and variabilities, their relations and the rules and dependencies among them. The activity of variability modelling aims at the definition of such a model using a specific variability modelling approach. The variability model can then be used to define product configurations by selecting valid variability combinations without any knowledge about the actual implementation. In this section, we discuss general requirements for variability modelling with a focus on the basic elements and capabilities that are typically needed to define and to use a variability model.

A variability model defines the valid **configuration space**[1] of a specific software product line. These variabilities are then implemented in the artefacts. The configuration space consists of a set of **atomic and configurable elements**. Each of these elements represents a specific variability. These basic modelling elements can be further described using additional elements, for example, configurable attributes for each basic modelling element. All elements of a configuration space are basically optional. In practice, the definition of constraints restricts this optionality, for example, to alternatives from which at least on element must to be selected or configured (we will discuss constraints in detail below). Determining a specific configuration of such elements, leads to the configuration of the corresponding software artefacts. In deliverable D2.2.1, we discussed several different approaches

[1] D5.1 acknowledges the general need for variability modeling and points to some specific aspects, which we will discuss in this and the following sections.

to variability implementation and the relation between model and artefact space. Some examples of such basic modelling elements are features in feature modelling [40, 65], variation points and variants in orthogonal variability modelling [58], and decisions and decision values in decision modelling [24, 63].

An illustrative configuration space may be the definition of the configurable elements of a car. We use a decision modelling notation for this definition. The decision model of a car DM_{Car} may be defined as a set of three decisions, namely the *driving system*, the *engine* and the *transmission*. The transmission further has a sub-decision *gears* that specifies the number of gears for a selected transmission (we use "."-notation to indicate a sub-decision):

DM_{Car} = {driving_system, engine, transmission,
transmission.gears}

The possible values of decisions `val(x)` are defined with respect to the available (physical) components of the modelled car. A customer can choose between a two-wheel or a four-wheel driving system, a gasoline or a diesel engine, and a manual or an automatic transmission. The transmission can have five to seven gears:

```
val(driving_system) = {two_wheel,four_wheel}
val(engine) = {gasoline,diesel}
val(transmission) = {manual,automatic}
val(transmission.gears) = [5,7]
```

A variability modelling language must support the definition of modelling elements to represent all variabilities in the artefact space. This includes the definition of modelling elements of different value types, e.g. enumerations (decision `driving_system`) or integers (decision attribute `transmission.gears`) and their possible values. We will further discuss details of the capabilities of variability modelling approaches in Section 3.

An application engineer uses such a variability model to define individual (software) **product configurations**[2]. A product configuration is a set of configured elements of the configuration space that represents exactly one product. The corresponding activity of defining a product configuration is called **configuration process**. The configuration of an individual car using the decision model DM_{Car} may lead to a product configuration $Conf_{Car1}$ specifying a four-wheel-drive, a gasoline engine and a manual transmission with six gears:

$Conf_{Car1}$ = (
driving_system = four_wheel,
engine = gasoline,
transmission = manual,
transmission.gears = 6
)

[2] D5.1, section 2.5.3: Platforms must be configured for different hardware and PLC configurations. D5.1, section 4.2.2: The result of the configuration process shall be a variability configuration model.

A variability modelling language must support the definition of multiple and individual product configurations on the basis of a variability model. The element combination defined in such a product configuration varies from product to product and, thus, has to be handled independently for each product configuration.

The definition of **constraints**[3] restricts a configuration space. These restrictions may be due to requirements, technical or implementation aspects. More precisely, the constraints restrict the combination of configuration space elements. These restrictions guarantee that the configuration of products using the variability model yield **valid product configurations**. A product configuration is valid only if the selected element combination does not violate any constraints. The intention is of course that a valid product configuration can be translated into a valid product implementation, i.e., results in a correctly working product. The configuration space of the car may be restricted to have a four-wheel-drive only in combination with an automatic transmission while an automatic transmission requires at least six gears:

```
driving_system == four_wheel requires
    transmission == automatic
transmission == automatic requires
    transmission.gears >= 6
```

Thus, the product configuration that we defined above would be invalid with respect to these constraints. The solution is to reconfigure the product (configuration) to either include an automatic transmission instead of a manual one or to change from four-wheel-drive to a two-wheel-drive.

A variability modelling language must support the definition of constraints to restrict the configuration space. This includes restrictions of the combination of configurable elements and complex dependencies among the individual elements (and their attributes). This capability is mandatory to provide a mechanism that guarantees the definition of valid and, thus, planned product configurations. The definition of constraints requires the support of different operator types, e.g. Boolean or relational operators (see the example above). The available operator types typically depend on the value types available for the definition of the configuration space.

The mapping of a valid product configuration to the artefact space leads to the instantiation of a valid product. Different approaches exist to realize such mappings. Schmid et al. [64] discuss different types of mapping between decision models and artefact space. Czarnecki et al. [17] have the same discussion on different mapping techniques for feature models. We described initially the mapping of valid product configurations in deliverable D2.2.1 as part of the variability implementation. We will continue this description in deliverable D2.2.2 based on the INDENICA variability modelling language defined in this deliverable. In the car example, the mapping and instantiation of the valid product configuration (we changed transmission to an

[3] D5.1, section 2.5.1: The configuration of the WMS system shall depend on the height and length of the high racks. The amount of high racks has a direct impact on the WMS. D5.1, section 2.5.5: The predefined relation between rack size and maximum weight per storage unit as well as between shape of the product, palette and warehouse specification shall be considered. D5.1, section 5.1.2: Restrictions on yard jockey and dock assignment strategies shall be considered.

automatic one) results in a real-world car with a four wheel driving system, a gasoline engine and an automatic transmission with six gears ready to drive.

The above discussion describes the general issues that are relevant when providing a variability modelling language, especially in the context of INDENICA. A list of the identified requirements can be found in Section 2.4 under "General Variability Modelling Requirements".

2.2 Variability Modelling in Service-Based Systems

In this section, we discuss specific requirements regarding variability modelling in service-based systems. Service Engineering has changed the development of software systems. A service-based system is no longer considered as an independent software product but provides local services and consumes other, maybe third-party services. Thus, it can be considered to be part of a service ecosystem. Variability arises naturally in these systems, e.g. as services can be easily exchanged or modified. This imposes specific requirements on the variability modelling language. In this section, we will discuss different aspects of variability in service-based systems and derive resulting requirements with respect to variability modelling.

In service-based systems, different **variability objects**[4] are relevant that may be impacted by a variability model. A variability object is the part of the service platform, service, or service-based application that is supposed to vary. The variability relevant to variability objects in service-based systems must be defined in a variability model. In deliverable D2.2.1, we identified and defined the following variability objects relevant to an INDENICA platform:

- **Service Platform Infrastructure:** This is the basic platform implementation, which cannot be further refined into specific services. This can be realized in an arbitrary (non-service-oriented) way. Variability in the basic platform implementation regards all basic functionalities, for example, user authentication mechanisms or persistency.
- **Technical Platform Services:** These are services that are provided from the technical platform. They enable functionality like the registration of services or other infrastructure capabilities. There can be variability regarding those, e.g., regarding the exact range of services or their exact behaviour.
- **Domain-Specific Services:** This includes any variability in domain-specific services where a service is modified, augmented by additional functionality, and otherwise adapted. In particular this may happen either while keeping the interface or modifying the service interface as well.
- **Service Composition and Processes:** This includes all cases where the specific composition of processes is modified. It encompasses in particular any situations where a specific service is explicitly exchanged for another service satisfying the similar interface, but behaving differently.

[4] D5.1, section 2.6 and 5.2.2.2: *All* services should provide variability to be tailored to the customer's needs.

- **Service and Platform Deployment:** This covers any form of variability that influences the specific deployment of a service[5] (e.g., not deploying, location of deployment, parameterization, etc.).

A variability modelling language for service-based systems and service ecosystems must support the definition of configurable elements appropriate to represent the above variability objects and their variability. In Section 4.4, we will further discuss possible solutions found in literature to represent these variabilities.

In deliverable D2.2.1 we also introduced different **forms of variations**. The following forms of variations[6] are rather generic, however, they get a specific meaning, as described below, in the context of service orientation:

- **Optional:** A variability object may only be part of an installation under certain circumstances. This is called optional variability. This might be a service, or a specific aspect of the functionality of a service or of the underlying platform.
- **Alternative:** Sometimes it is important that one of several variability objects is present, but the variation is in which of the objects to pick. This can be, for example, one of several possible service realizations that adhere to the same interface or alternative behaviours of a platform infrastructure.
- **Multiple selection:** Sometimes multiple options from a set of variability objects can be selected.

A variability modelling language for service-based systems and service ecosystems must support at least the definition of the above forms of variations. The definition may either affect only a single variability object (e.g. an optional variability object) or groups multiple variability object[7] (e.g. a set of alternative variability objects). Thus, in case of alternative and multiple selections, these forms of variations require the definition of value ranges or arrays including cardinalities to represent the set of selectable variants and the number of possible selections. Alternative and multiple selection also require grouping of modelling elements, e.g. to define a set of alternatives of services from which the selected variant must be configured individually. The definition of arrays and the grouping of configurable elements also support large-scale variability by organizing related variabilities into manageable units.

Other important forms of variability in service-based systems are parameterization and extension[8]. Parameterization may require the definition of an open-ended set of parameters. Many existing variability modelling languages also provide a concept of parameterization. However, in these cases there is typically a fixed set of possible

[5] D5.1, section 4.2.2: Generation of deployment descriptors and other configuration parameters based on product configuration.

[6] D5.1, section 5.2.2: In the YMS case, the persistency mechanism is a classical optional selection while connectivity and authentication are multiple selections. However, configuring an entire WMS topology (D5.1, section 2.5.1) needs at least all three forms of variation.

[7] D5.1, section 2.5.1, 2.5.5, 2.5.6: Needed to describe warehouse topology. D5.1, section 5.2.2: Needed to group variabilities for individual subsystems.

[8] Interface variability was also mentioned in D2.2.1 as a possible form of variation, however, we regard this mainly as a form of variability implementation (as was appropriate in D2.2.1). In terms of modelling interface variability this can be mapped to the forms of variability identified above.

parameters. Thus, the addition here is strongly related to the next point: the concept of extension. Extension[9] means that new variations can be supported that are not known at the point in time of defining the variability model. This is a typical aspect of service-based systems and hence must be adequately supported. It is also strongly related to the aspect of ecosystems, which we will discuss below.

Quality in service-based systems is typically specified as **Service Level Agreements (SLA)** in terms of **Quality of Service (QoS)**. QoS-requirements can be seen as a kind of constraint on the behaviour of a service that prescribes actions and/or states of this service that a service provider accepts and advertises to service consumers [34]. QoS may be expressed in terms of qualitative statements or quantitative (numerical) statements. One example of QoS is the availability of a service, which may vary according to the concrete selection of a service variant. For service availability a qualitative statement may be “high” while a quantitative statement could be 99 percent. A SLA is a formally negotiated agreement between a service consumer and a service provider or between service providers that transcripts the common understanding about services, their priorities, responsibilities and service guarantees concretizing individual QoS statements [25]. A concrete SLA may specify that the QoS availability for a given service has to be greater than 90 percent.

A variability modelling language for service-based systems and service ecosystems must support the specification of qualitative and quantitative properties[10] describing QoS for individual configurable elements. Especially for the treatment of quantitative properties arithmetic expressions[11] are useful. An example could be the overall quality of a service to be computed from responsiveness and availability. Constraints on QoS properties[12] must be supported to express valid ranges and dependencies among QoS. Constraints may be used (in combination with arithmetic expressions) to map between qualitative and quantitative properties.

Integration and configuration of service platforms are core topics of the INDENICA project for which in a concrete setting a large number of decisions has to be made. Examples of such decisions are which technologies should be used for integration or which services should be made available at which point in time (through configuration). Making such decisions at a certain point of time during the software development cycle typically fixes several cornerstones of a service-based system which cannot be changed later. Two particular types of such decisions are 1) the **service technology** used for integration which is in a concrete implementation typically interwoven with the functional code of the services and 2) the point in time

[9] D5.1, section 2.5.3: Upgradability and extensibility of a WMS within a fixed timeframe.

[10] D5.1, section 2.5.2: QoS such as latency and throughput may determine some variabilities of a WMS. D5.1, section 3.2.2: Minimal latency is essential for a WMS. Availability may be a specific variability in a WMS. D5.1, section 4.1.1: Remote Maintenance Systems are created for environments with specific QoS requirements which, thus, may impact the configuration of such a system.

[11] D5.1, section 2.5.2: Relation between size of goods and speed of conveyer system or speed of communication system may be expressed in arithmetic formulae or as constraints. D5.1, section 2.5.4: Runtime variability which depends on the rate occupancy indicator.

[12] D5.1, section 2.5.2: The type of goods handled by a WMS may influence the specific distances and, thus, latency and throughput. D5.1, section 3.2.2: The operator may switch to high-availability. This leads to the activation of AppFabric’s built-in high-availability caching option.

when all decisions about a configuration space element must be made, the so called **binding time**[13]. Service technology and the range of possible binding times may be expressed as aspects of a configuration space element. Enabling the variation of these two aspects greatly increases the flexibility of integrating and configuring service platforms. We call this **meta-variability**, i.e. the systematic variation of aspects of a variability which are usually considered as fixed.

A variability modelling language for the integration and configuration of service-based systems must support the definition of meta-variabilities and their individual properties as well as the impact of meta-variabilities, i.e. to bind them as part of the configuration process. In addition, the restriction of meta-variability requires constraints, e.g. to express that configuring the binding time of one configuration space element restricts the binding time of a dependent configuration space element.

Special requirements arise for variability modelling in service ecosystems. In the context of the INDENICA project we can extend the notion of service ecosystems to **service platform ecosystems** in which multiple different service platforms (or parts of them) form an integrated and domain-specific platform. The characteristics as well as the requirements for variability modelling in service ecosystems can be transferred to service platform ecosystems. In service (platform) ecosystems each platform holds its own variability model. The configuration of a domain-specific platform requires the **composition** of these models[14]. The composition mechanism must guarantee both the individual configuration of each (sub-) platform and the valid configuration across platform boundaries. The latter requires the definition of constraints among modelling elements of different variability models, which in turn requires a clear identification of each modelling element with respect to its source (the variability model of the respective platform). In addition, service platform providers may pre-configure some variability, e.g. the number of services that a specific customer is allowed to deploy. This results in a **partial configuration**[15], which must remain over the course of integration, and further configuration of the overall domain-specific platform.

A variability modelling language for service platform ecosystems must be modular and extensible to support the composition of multiple variability models as well as their individual configuration and the configuration across variability model boundaries. This facilitates the definition of constraints among modelling elements of different variability models. Each modelling element must be clearly identifiable with respect to its source (variability model). The modelling language also must support partial configuration to enable pre-configurations of modelling elements.

[13] D5.1, section 5.1.3: Platform bundles as a way of implementing variability can be deployed, updated and removed (even) at runtime. D5.1, section 4.2.2: Deployment descriptors are considered as one means of variability, thus also deployment time binding is required.

[14] D5.1, sections 4.1.4 and 5.1.4: This may be part of the activities for combining the base platform with several domain services and their individual variability models.

[15] D5.1, section 2.5: Combining and integrating base platforms to a whole VSP.

The requirements we identified in this section serve as a basis for a detailed discussion on advanced variability modelling concepts in Section 4. In this section we will analyse various options of addressing these requirements. Section 2.4 contains a full list of all requirements under "Requirements for Variability Modelling in Service-Based Systems".

2.3 Requirements from Industry

While the previous subsections took a rather fundamental point of view with respect to variability, this section emphasizes the industrial perspective. We gathered requirements from the various industrial partners and used this as a basis for evaluating the analysis given in the preceding sections and summarized in Section 2.4. By the very nature of this approach, the identified requirements overlap with the ones described so far. However, in this subsection we will emphasize the additional ones.

In terms of basic modelling facilities, the importance of having more modelling elements than only Boolean variability availability has been unanimously raised. Thus, enumerations[16], numbers[17], or strings[18] can be regarded as a must. The motivation for this was sometimes based on the need to represent Quality of Service characteristics, but other aspects like business constraints also played a role.

The replication of features (feature cardinality) and constrained selection of elements within a group (group cardinality) [20] are also mentioned as important. In particular, the need to support the replication of sub-configurations[19] has been repeatedly mentioned (corresponds to feature cardinality).

Other aspects that impact the required basic expressiveness of the modelling approach are:

- The capability to reference as part of the modelling language other (sub-) configurations explicitly and thus to express references to entities explicitly.
- The capability to integrate higher-level configuration and generation capabilities (e.g., in the form of DSLs) into the approach.

Some of the requirements are also related to the expressiveness of constraints. Especially the need for being able to use high-level (abstract) requirements to configure more detailed aspects and the need for numerical (computational) constraints was repeatedly emphasized. The need for numeric constraints was partially driven from the need to address Quality of Service aspects, but also partially from the business aspects. In the context of multiple binding times, it was also

[16] D5.1, section 4.2.2: Examples for the Remote Maintenance system are different databases or communication channels. D5.1, section 5.1.4: Examples for a YMS are the yard jockey state and its position.

[17] D5.1, section 2.5.1: For example size of the warehouse, amount of high racks, height and length in storage units. D.5.1, section 2.5.5: Size, weight and shape of products in a warehouse. D5.1, section 5.1.4: Examples for a YMS are the number of docks or the number of goods.

[18] D5.1, section 4.2.2: See the detailed options of the (pre)configuration of the Mobicents platforms as well as server and communication settings of WMS or YMS servers.

[19] D5.1, section 2.5.1: Cardinalities and restrictions of multiple selection are in particular relevant for specifying and configuring WMS topologies.

emphasized that constraints might be binding-time-dependent and, thus, require binding-time-specific treatment.

Some of the requirements that were brought up also concerned the ease of performing configurations in practice. However, we expect that some of these aspects can be handled through adequate tool support, independent of the specific modelling approach. Thus, they do not need to be part of the variability management approach per se. We summarize them here for the sake of completeness:

- The need to be able to provide views (according to different criteria) of the variability model to reduce the complexity a user has to deal with.
- The need was raised to work with defaults (e.g., in the form of business profiles) to simplify the task of configuration.

Another category of requirements that was mentioned on various levels is the need to be able to deal with several variability sub-models. This mostly overlaps with the issue of ecosystems, which we addressed above. This issue was raised in several forms:

- The ability to deal in an integrated manner with variability models that cover multiple sub-systems[20] (e.g., different types of services that come from different backgrounds).
- The need to deal with the integration of legacy product lines, in particular, to interface with their configuration processes.
- The need to decompose complex variability models to a set of sub-models that can be handled more or less independently.

Some requirements were also more technological in nature. These concerned the alignment of different technology layers or components to achieve a well-orchestrated realization of the variation and the need to be able to influence specific configuration aspects like technical platform services.

We will summarize the resulting requirements in the following section and also outline the overlap with the requirements that we identified so far.

2.4 Summary

In this section, we discussed what we regard as key requirements for the variability modelling approach in the INDENICA project. We gathered those requirements from discussions of general demands in variability modelling, the investigation of demands for variability modelling in the context of service-based systems and service ecosystems, and the information from the industrial partners on their current (and expected future) situation. We will now summarize the identified requirements for future reference[21]:

[20] D5.1, section 4.2.2: For example subsystem variants in the Remote Maintenance system.

[21] Throughout the description of the requirements, we omit the phrase "the variability modelling approach should support the... " for the sake of simplicity and readability.

General Variability Modelling Requirements

G1 Definition of a configuration space that represents all variabilities as a composition of basic modelling elements.

G2 Definition of additional modelling elements (attributes) that may refine the basic modelling elements and allow representing additional aspects of variability.

G3 Definition of different types of value ranges, including non-Boolean variability.

G4 Definition of product configurations, that allow describing individual configurations based on the definition of the configuration space.

G5 Definition of constraints to restrict the combination of elements defined in the (unconstrained) configuration space.

G6 Definition of complex dependencies among the individual variability elements (and their attributes).

Requirements for Variability Modelling in Service-Based Systems

S1 Definition of the variability relevant to service platform infrastructure, technical platform services, domain-specific services, service composition and processes and service and platform deployment. This requirement is an extension of the general requirement G1.

S2 Definition of (at least) the following forms of variation: optional, alternative and multiple selection.

Large Scale Variability[22]

S3 Grouping of related variability elements, e.g. to define a set of alternative services from which the selected variant must be configured individually. This requirement is an extension of the general requirement G1 and helps to organize large scale variability.

S4 Definition of "arrays" to represent sets of cases that need to be configured while the relevant configuration possibilities have the same structure.

S5 Definition of strong dependencies in the sense that general (abstract) decisions lead to the configuration of (multiple) lower-level decisions.

Quality of Service

S6 Specification of qualitative and quantitative properties describing QoS for individual configurable elements.

S7 Specification of arithmetic expressions to specify derived quality properties (an example could be the overall quality of a service to be computed from responsiveness and availability).

S8 Definition of Constraints on QoS properties to express valid ranges and dependencies among QoS. Constraints may be used (in combination with

[22] Further requirements for large-scale development can be found under eco-systems, as they are specific to the eco-system case.

arithmetic expressions) to map between qualitative and quantitative properties. This requirement extends the general requirements G5 and G6.

Meta-Variability

S9 Definition of meta-variabilities and their individual properties. This requirement is an extension of the general requirements G1 and G2.

S10 Definition of the impact meta-variabilities have on the implementation process. This requirement is an extension of the general requirement G4.

S11 Definition of constraints on meta-variabilities, e.g. to express that configuring the binding time of one configuration space element restricts the binding time of a dependent configuration space element. This requirement is an extension of the general requirements G5 and G6.

Eco-Systems

S12 Composition of multiple variability models as well as their individual configuration and the configuration across variability model boundaries.

S13 Definition of constraints among modelling elements of different variability models including a clear identification of each element with respect to its source.

S14 Partial configuration of variability models to enable pre-configuration and reuse of existing variability models in new contexts.

S15 Extension of the configuration space by variabilities that have not been taken into account previously (open variation). This can be seen as a special case of S12, however, it goes beyond it by demanding that existing variations can also be extended at a later point.

S16 Separation between local and global variability implementation reusing variability models (modularity). In particular, this leads to the requirement of variability interfaces.

Requirements from Industry

I1 Definition of non-Boolean variability. This is subsumed by requirement G3.

I2 Definition of cardinality (i.e., making – restricted – choices in groups of elements and replicate complete groups in order to provide individual configuration possibilities for the various copies. This can be seen as an extension to the requirement S3.

I3 Referencing of other (configuration) elements.

I4 Integration with existing domain-specific languages. Due to its specialized nature, we consider this as a low-priority requirement, but analyse this further in Section 3.6.

I5 Automatic deduction of lower-level configuration choices from higher-level configuration selections. This is subsumed by requirement S5.

I6 Definition of numerical constraints. This is subsumed by requirements G6, S7, S8.

I7 Definition of binding-time dependencies of constraints.

I8 Definition of different views (according to different criteria) of the variability model to reduce the complexity a user has to deal with.

I9 Working with defaults, including complete default profiles describing defaults. This may be handled as well on the tool level.

I10 Integrated configuration of modularized (and composite) product lines. This is mostly subsumed by S12 and S15.

I11 Integration with legacy product lines. In particular, the harmonized configuration of different product lines. This is mostly subsumed by S12 and S15.

The identified requirements serve as a basis for our discussion on core variability modelling concepts in Section 3 and advanced variability modelling concepts in Section 4. In addition, we will reconsider these requirements in our description of the INDENICA variability modelling approach, which we will define in Section 5.

3 Core Variability Modelling Concepts

The purpose of this section is to provide an overview of general variability modelling concepts, which we will use as a basis to select the core concepts of the INDENICA variability modelling approach. We will describe different categories of expressiveness of modelling concepts. The expressiveness of the concepts will increase over the course of this discussion. Our focus will be on the modelling elements introduced by these categories and the different constraint and operator types available for dependency management. We will further include existing approaches that use the specific concept and compare the benefits and drawbacks that come with the respective expressiveness. A running example will illustrate the various levels of expressiveness in variability modelling throughout these sections. We will start with an introduction of this example in Section 3.1.

The following sections will discuss five different levels of expressiveness from less powerful to more powerful. Of course, different languages cannot exactly be put in different levels and matched with different expressiveness of constraints. This would actually form a two-dimensional matrix. Thus, our definition of levels is an approximation.

3.1 Running Example

In this section, we introduce a running example which will be used throughout Section 3 to illustrate the various levels of expressiveness in variability modelling. The example will evolve with each level of expressiveness discussed in the following sections.

The example will model the variability of the instantiation and deployment of a content-sharing application. A content-sharing application allows its users to upload, annotate, release and share content of various types. In this example, concrete applications may differ with respect to:

- The *supported content types* such as text, video, audio, 3D content, or binary (large) objects (BLOBs).
- The *hosting infrastructure* which consists of a) a web container being responsible for serving the content and b) the database, which stores user and content data.
- The *deployment target*, which may either be a traditionally hosted server or a cloud environment. The cloud environment may be private, like a local installation of the Eucalyptus[23] cloud software or public, in this example we will allow from the Amazon[24] or Azure[25] cloud.

[23] http://open.eucalyptus.com/

[24] http://aws.amazon.com/de/ec2/

[25] http://www.microsoft.com/windowsazure/

Without going into functional details of the content-sharing application, the variabilities introduced by content types, web container, database and deployment target allow to derive a large number of different application instances. Whether we can describe a specific set of applications at all or whether it is possible to state individual details depends on the expressiveness of the languages used for modelling variabilities and dependencies. In the following subsections, we will successively augment the modelling of the running example with the restrictions and dependencies below:

1) At least one content type must be present as otherwise the content-sharing platform is useless.
2) To ensure acceptable quality of service, the maximum bit rate for video content on the Tomcat web container is 128 kBit/s.
3) The combination of supported content types may be restricted based on the capabilities of the web container or the deployment platform, e.g. due to load problems only a limited number of content types may be available on the traditional deployment target.
4) Some content types may be served by a separate web container in order to configure a simple load balancing mechanism, for example 3D content should be served by a JBoss server. As a further extension, a web container may be configured to retrieve its content from a specific database.
5) Content types may be transformed and the result may be shared. Such transformations should be configured in terms of configuration chains, such as the textual representation of the audio track of a video. As transformations may be resource-consuming and, thus, affect the performance, on the traditional platform only simple and resource saving implementations should be deployed while resource-consuming high-quality transformations may be used on the cloud platforms.

The concrete notation used for illustrating the individual versions of the running example will be explained as part of the discussion of the levels of expressiveness in the following subsections.

3.2 Basic Variability Modelling

The simplest language, we will discuss, is a purely Boolean representation of variability elements. This is the case in basic feature modelling. Also on this basic level, we will restrict ourselves to binary constraints where one feature can be mutually exclusive with another one or may require another one.

Feature diagrams were first introduced as part of the Feature Oriented Domain Analysis (FODA) feasibility study in [40]. The intention of this method is the identification of prominent or distinctive features. These features are attributes of the system that directly affect end-users. They are user-visible aspects or characteristics of the domain. The features define both common aspects of the domain as well as differences among related systems in the domain. Thus, feature modelling is a kind of variability modelling, whereas feature diagrams serve as a communication medium between users and developers.

Meanwhile, there exist a plethora of feature modelling approaches, each developed for a specific purpose. At this point, we dispense with a detailed analysis of existing feature diagram notations as this can be found in literature, e.g., in [65]. This semantics also forms an input to our discussion of variability modelling described here.

Feature diagrams consist of mandatory, optional and alternative features. They are typically hierarchically organized, thus the selection of a sub-feature requires the selection of its parent feature(s). Unlike FODA features diagrams which are trees, today's feature diagrams are in general single-rooted directed acyclic graphs (DAG), because they represent graphically requires and mutually-exclusive relations among features. However, newer approaches are not necessarily more expressive than FODA. The FODA approach is capable of handling non-Boolean features like "Horsepower", while many feature diagram notations can only handle Boolean features. Further, FODA makes use of textual "Rationales" to support the selection process.

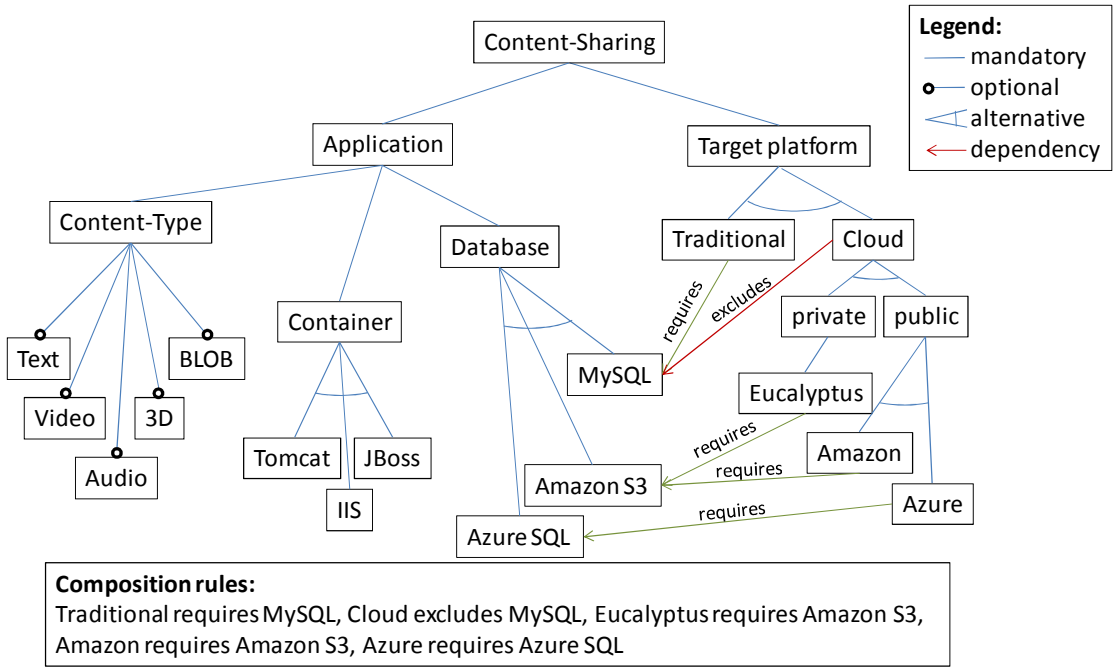

Figure 1: Running example using basic feature modelling.

Figure 1 depicts the running example as a basic feature diagram. The content-sharing application consists of an application part and a target platform part. For the application the available content types are modelled as optional features, i.e. each of the content types as well as arbitrary combinations may be selected. For the container one may choose among the alternatives `Tomcat`, `JBoss` and `IIS` (Microsoft Integrated Information Server). Similarly, the database may be `MySQL`, `Amazon S3` or `Azure SQL`. The target platform part is either `Traditional` or `Cloud`. Cloud is further decomposed into `private` clouds (`Eucalyptus`) and `public` clouds (`Amazon` or `Azure`).

The composition rules shown below the feature diagram restrict the set of valid configurations. While the `Traditional` target platform requires `MySQL` (this implicitly excludes the other alternatives), for `Cloud` target platforms `MySQL` must

not be selected as it is excluded. Eucalyptus and Amazon require the selection of Amazon S3 (excluding the other alternatives) while Azure requires Azure SQL. Additionally, we visualized the composition rules as dependencies in the feature diagram (either textual rules or dependency arrows are sufficient while visual dependencies are not part of the original FODA notation).

This basic feature modelling approach is not sufficient to model the restrictions listed in Section 3.1. In particular a content-sharing platform is useless if no supported content type is selected. Modelling such situations requires cardinalities, which we will discuss in the next section.

3.3 Cardinality-Based Variability Modelling

In this section, we introduce cardinalities to the previously described modelling approach. Two different types of cardinalities can be distinguished: feature cardinalities and group cardinalities. Feature cardinalities facilitate multiple instantiation of a feature, whereas group cardinalities describe how many sub-features of a group of features can be selected.

The two concepts can be combined. This kind of grouping allows the multiple selection of sub-features. This new grouping mechanism can be treated as a kind of group cardinalities. These cardinalities specify a minimum and maximum number of sub-features, which must be selected. Thus, alternatives can be modelled with <1-1>, or-grouping with <1-n> group cardinalities.

A detailed analysis of these extensions can be found in [18].

Cardinalities can be seen as a specific kind of constraint. Besides this, no other type of constraint directly "belongs" to this level. However, we will also introduce on this level arbitrary Boolean expressions as a basis for constraints. This is obviously an extension of requires and excludes relations, which can always be written as a binary Boolean relation of the form $A \rightarrow B$ (A requires B) or $A \rightarrow \neg B$ (A excludes B). Thus, arbitrarily complex Boolean constraints actually provide a generalization (they might be simulated using requires and excludes, however).

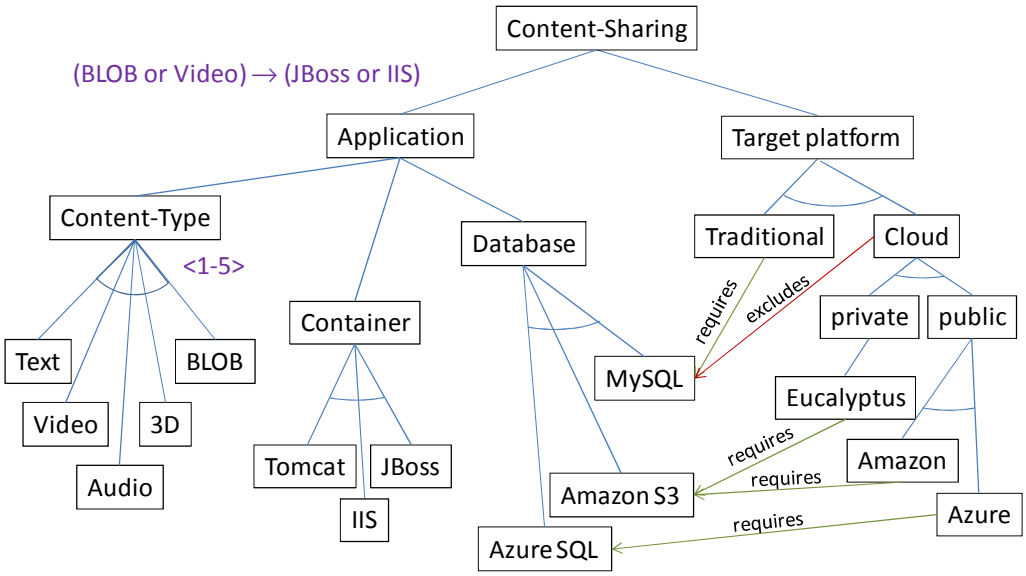

Figure 2: Running example using cardinality-based feature modelling.

Figure 2 shows the extended running example with a feature group for content types and a cardinality, which restricts the number of selected options for a valid configuration, here that at least one content type must be selected. Further, the Boolean constraint `(BLOB or Video) → (JBoss or IIS)` restricts that if either a `BLOB` or `Video`-content is used than as container `JBoss` or `IIS` must be used. Note, that in this special case, this can also well simulated with requires and excludes relations, but our main point is the emphasis that in general people can express their intentions more precisely. While now arbitrary Boolean constraints (which encompass inclusions and exclusions) may be expressed, it is not possible to state a basic QoS constraint such as: when selecting `Tomcat` the bitrate of a `Video` is at maximum 128 kBit/s. We will discuss such non-Boolean constraints in the next section.

3.4 Non-Boolean Variability Modelling

The example in the previous section showed that it would be good to be able to describe variability not only in terms of Boolean variability, but also in other types like integer, string, etc. This is summarized as non-Boolean. In [56], the authors discuss that non-Boolean variability plays an important role in practice. In this section, we will first introduce the concept of non-Boolean expressions in variability modelling and the respective constraint mechanisms, then we discuss some approaches and illustrate the use of non-Boolean expressions in our running example.

In variability modelling, the definition of non-Boolean elements, like a configurable element that assumes integer values, either requires the definition of a type or the exact (range of) values that the element may assume. In both cases, each value represents exactly one variant of a variability of the artefact space. Constraints restrict the selection of these values to yield valid value combinations. The definition of these constraints requires additional operator types, such as relational, arithmetic or string operators. The operator types typically depend on the element types that a variability modelling approach supports.

Non-Boolean elements and expressions are basic modelling elements in decision modelling approaches [64]. Schmid and John [63] as well as DOPLER [24] support decision types like Boolean, string, enumeration, integer, real, and additional sub-ranges as well as set-based data types. Synthesis [67] and VManage [28, 46] also support decision types like date and time. In [9], the authors discuss that non-Boolean elements extend basic feature modelling, e.g., to define cardinalities among feature sets or additional attributes to describe properties of a feature. Different feature modelling approaches exist that support these extensions and the definition of constraints among them [6, 7, 8, 20, 21, 39, 69, 74].

Non-Boolean elements and expressions enable the definition of multiple values for each configurable element which yield a fine-grained structure of configuration spaces. The definition of constraints not only restricts the combination of configuration space elements but also facilitate the calculation of element values by using arithmetic operators. The major drawback of non-Boolean expressions is their analysability as Boolean expressions are NP-hard and non-Boolean expressions extend this complexity to be undecidable [56].

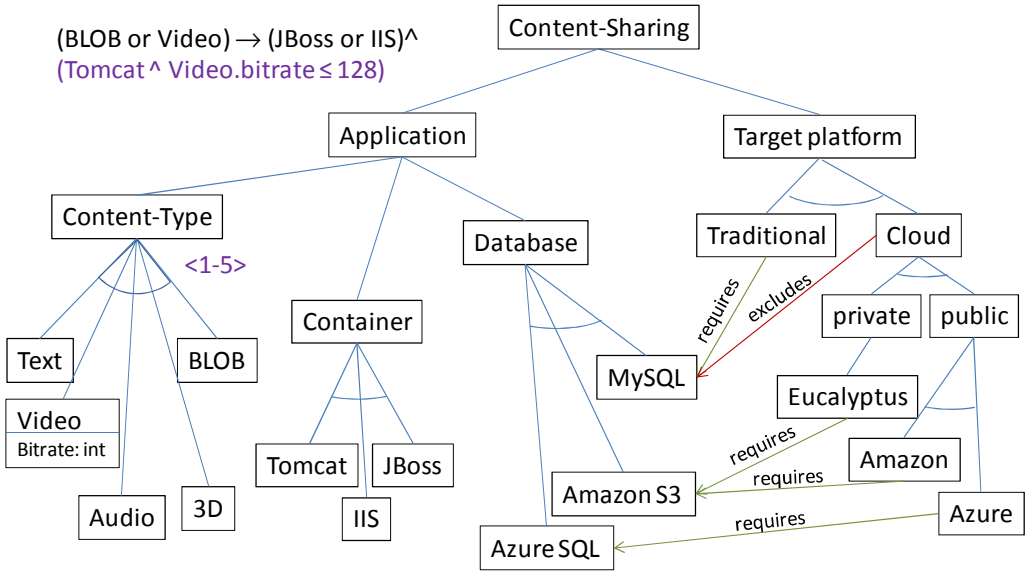

Figure 3: Running example with feature attributes and non-Boolean constraints.

Figure 3 depicts the extended variability model in order to express a non-Boolean QoS constraint restricting the maximum bitrate of videos provided by a certain web container. The `Video` variability is specified by the integer attribute `Bitrate` which is used in the non-Boolean constraint relating `Tomcat` to the maximum video bitrate of 128 kBit/s. Note that the variability model does neither specify the origin of the value nor the binding time of the bitrate, i.e. the value might be provided during configuration prior to runtime but it might also be left open as a runtime variability to be obtained from a runtime monitoring service and to control runtime reconfiguration of deployed web containers.

The constraints discussed so far represent expressions on the elements used in the variability model such as variabilities, attributes or dependencies. A further (minor) increase of the expressiveness could allow **expressions on meta-model information** such as constraints on the cardinality of a variability. Let `a.card` be the cardinality

of the variability a. Then we can model that for an instantiation of the content-sharing application to be deployed to a `Traditional` target platform at most two content types may be selected:

$$\texttt{Traditional} \wedge \texttt{Content-Type.card} \leq 2$$

The delivery of some of the content types such as 3D or BLOBs may be resource intensive and impact the performance of the entire content-sharing application. An approach to improve performance could be the following simple load balancing mechanism: Serving the resource intensive content by separate subordinate web containers which are deployed to different physical machines or virtual cloud instances. Then, a request to such content would be delegated from the main application web container to the subordinate web container. Modelling this load balancing mechanism as configuration options using the variability modelling approaches discussed so far, we would need to replicate the container variability as a decomposition of 3D and BLOB. This is illustrated in the model fragment shown in Figure 4 where replicated sub-trees are highlighted as gray areas.

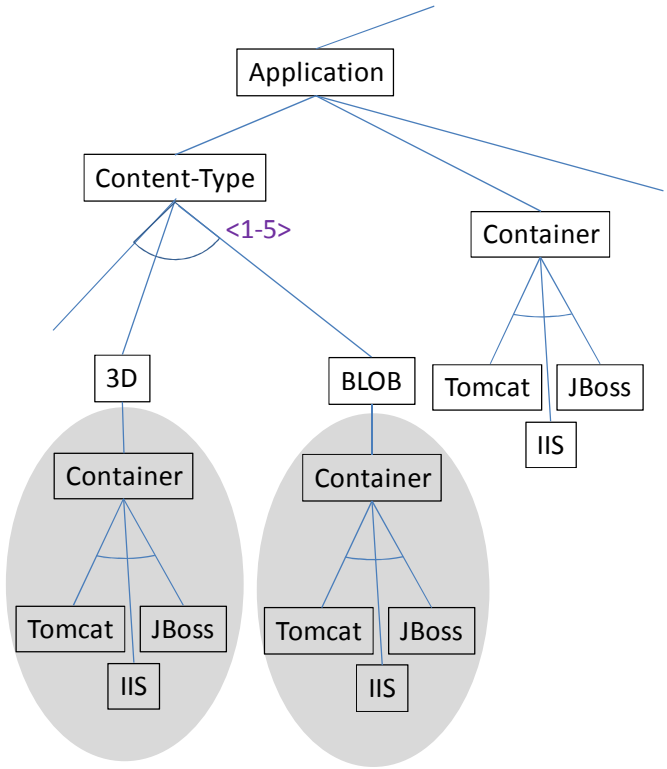

Figure 4: Replicated sub-trees in the running example (model fragment).

Feature cardinalities as introduced in [19] can be considered as a modelling alternative to reduce the number of replicated sub-trees. Figure 5 depicts a modification of the running example in which multiple instances of the content type feature can be defined. Each instance of the content type feature is flagged by an individual string indicating the selected content type, such as `Content-Type("BLOB")`. At a first glance, this modelling alternative seems to reduce the number replicated sub-trees but it also implies that each (and not only selected content types) may be served by an individual container. Further, the modelling in Figure 5, which is aligned to the examples in [19], defines an unnecessary openness

stated in terms of the string attribute, i.e. not only strings representing valid content types may be used. This may be avoided using enumeration types.

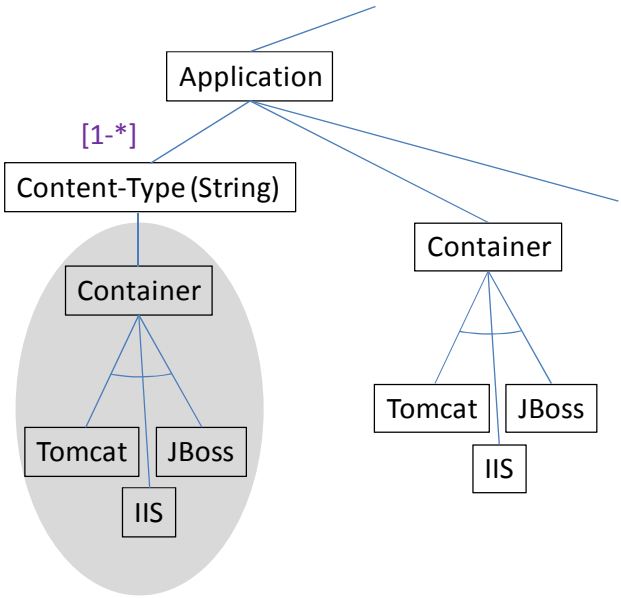

Figure 5: Replicated sub-trees using feature cardinalities (fragment).

However, in both cases an extension of the variability modelling capabilities by configuration references can help solving the problem of replicated sub-trees.

3.5 Configuration References

In this section, we introduce the concept of configuration references. In the first part, we explain configuration references, the elements to define such references and the corresponding dependency management. We will discuss existing approaches that include configuration references in their variability modelling concept. Finally, we will illustrate configuration references using the running example.

A configuration reference is a link from a configurable element *A* to a configurable element *B* specifying that the configuration capabilities defined by *B* become available in the context of *A*. Primarily, linked configuration capabilities are made available in an exclusive form, i.e. during the configuration process individual values can be specified for each source of a configuration reference. However, linked configuration capabilities may also be shared (as e.g. discussed in [4]). The concept of referring to and, thus, reusing configurable elements is similar to object references in object-oriented languages. There, (visible) capabilities of a class become available through the object reference. The target of the object reference may be a shared or an individual instance and even the reference may be shared (static).

By defining a configuration reference, the constraints for referred configurable element *B* naturally apply to the configuration capabilities, which become available in the source *A*. Further, *A* may define constraints to restrict the use of the referred configuration options within its own context. Additionally, the constraint language may be augmented to provide specific operators for configuration references. For

example, in case that the variability modelling language supports inheritance of configurable elements, a *typeOf* operator may constrain the concrete subtypes of *B* which may be used in the context of *A*.

Bak et al. include configuration references in their Class Feature Relationships (Clafer) approach [4]. The authors use abstract Clafers as a way of reusing parts of feature models. These abstract elements cannot be selected or deselected as part of a product configuration, unless they are extended by a concrete element. References then define relations to such concrete elements while mapping constraints define their individual configuration. Shared targets may explicitly be forbidden for individual references in Clafer. Reiser introduces configuration links in his Compositional Variability Management framework (CVM) [59]. CVM reduces the complexity of real-world feature models by decomposition. A feature model in CVM, thus, may be a composition of multiple (sub-) feature models. Configuration links between (sub-) feature models allow the definition of how to configure a feature of a target model depending on the given configuration of a source model. Boucher et al. offer the definition of custom variability types that can be reused in a variability model based on their Text-based Variability Language (TVL) [12]. Custom variability types factor out recurring variable elements. Each feature that includes an element of this custom type is able to configure its variant individually.

The benefits of configuration references in variability modelling are the reduction of complexity and the relation between multiple, individually configured elements of a configuration space. Configuration references reduce complexity by reusing configurable elements in the configuration space. In addition, each new reference allows the configuration of an individual instance that is exclusive for the referring element. The major drawback of configuration references is their analysability. Similar to non-Boolean expressions we introduced in the previous section, variability models including configuration references are undecidable.

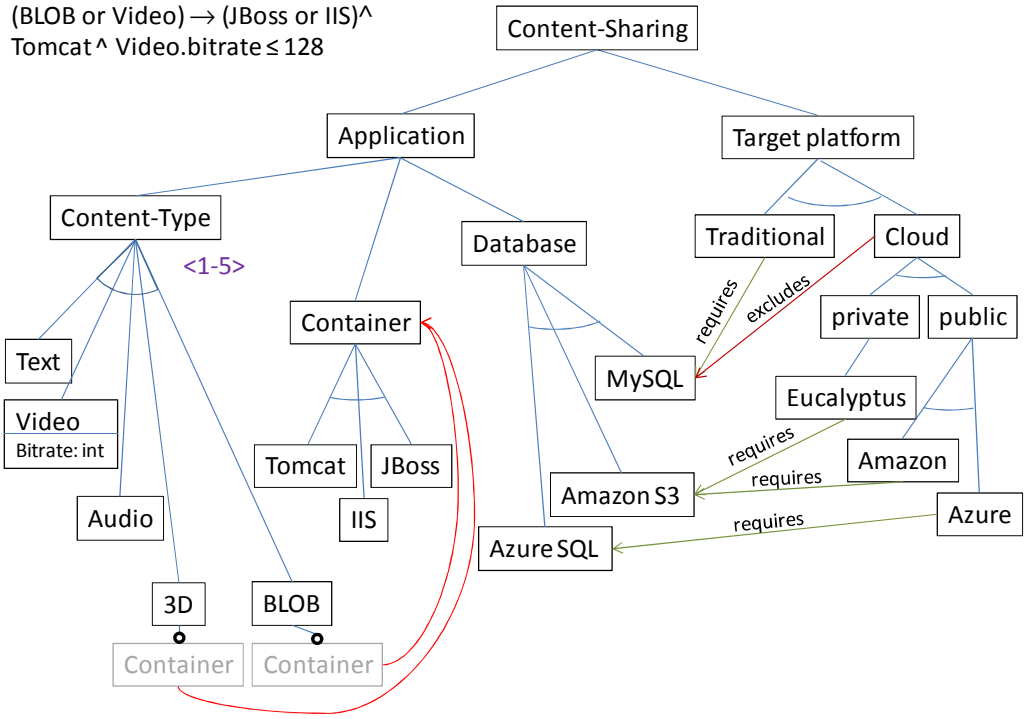

Figure 6: Running example with configuration references.

Figure 6 shows configuration references to specify the configuration of the simple load balancing mechanism introduced in Section 3.4. We apply the notation from [73], i.e. we denote configuration references in light grey. Additionally, we highlight the references in Figure 6 as additional arrows. The variability model in Figure 6 specifies now that 3D content or BLOBs may have individual web containers, which are defined by the `Container` introduced as a decomposition of `Application`.

While configuration references provide more expressiveness than the previous levels of description, there are still exist cases that cannot be fully described with this expressiveness. Thus, sometimes the need for using domain-specific languages also for variability modelling languages is voiced [73]. We will address this case in the following section.

3.6 Domain-Specific Languages

The idea of a domain-specific language (DSL) as discussed in [29, 32, 33] is to target a particular kind of problem in an intuitive way using concepts and elements, which are well-known in the problem domain rather than general-purpose concepts. In this section, we will discuss options using DSLs in variability modelling, which we identified from literature. We will also illustrate the use of DSLs in the context of our running example.

We identified the following strategies for using DSLs in variability modelling:

1. **Use an existing domain-specific language** and exploit some of the already defined elements to realize variability modelling.
2. **Define a domain-specific variability language** to provide means for expressing variability and product configurations in a problem domain.

3. **Extend a general-purpose language** by a DSL which provides domain-specific variability modelling concepts.
4. **Extend a domain-specific language** by concepts taken from a general-purpose language to increase the expressiveness of the DSL.
5. **Combine languages,** in particular of DSLs, to improve reuse of existing concepts in a more focused language.
6. **Instantiate a DSL** from a meta-model, e.g. a meta-DSL, which describes a family of variability-enabled languages. This leads to languages with a common core, e.g. to provide common mechanisms for traceability.

The two main approaches for realizing the extension and combination of languages are (a) to derive a new combined language (called amalgamated approach in [36]) and (b) to apply loose coupling, e.g., using links (called separate-language approach in [36]).

Several concrete approaches using DSLs for variability modelling are described in literature. We will discuss some selected approaches in the following according to the sequence of strategies listed above. Matlab / Simulink is a widely used tool in the embedded systems domain. The authors in [10, 11] use Matlab / Simulink as an *existing domain-specific language* and map concepts from a Matlab / Simulink model to a variability model at the beginning of domain engineering and derive a configured model as result of application engineering for final processing by Matlab / Simulink. In [73] the authors discuss *specific DSLs* for describing the product configuration of a fountain and an alarm system menu product line. In this work, the authors exploit hierarchical nesting of DSL elements to constrain valid product configurations. FAMILAR, as described in [1], is a domain-specific language for feature modelling which particularly supports inter-model manipulations such as merging. In [49], the authors discuss the *extension of a general-purpose language by a DSL*. There, the authors extend the Meta Object Facility (MOF) [51] by variability constructs using aspect model weaving. In several approaches such as [16, 35, 36, 75, 76], the Unified Modelling Language (UML) [53] as a general-purpose modelling language is extended with variability modelling concepts. The reversed case, i.e. *extending a domain-specific language by a variability modelling language* is discussed in [36] where the authors enrich a train control language by their common variability language (CVL) [55].[26] The combination of several DSLs for variability modelling is discussed in [73]. The authors combine one DSL for expressing the logical structure of a product with a second language for specifying behavioural aspects of the product. The application of a *DSL as a language for describing DSLs* is the topic of [77]. In this work, the authors describe VML*, a family of languages for variability management to derive variability-enabled languages for the entire software development lifecycle exemplified by a requirements and architecture modelling language.

[26] The CVL proposal is available to OMG members via the OMG website; non-members can obtain it, on request, from Oystein Haugen, the proposal editor.

Using DSLs in variability modelling requires also an appropriate constraint language, i.e. concepts representing variability in a DSL as well as their properties must be accessible for expressing constraints on them. Most of the publications discussed above do not give details about the applied type of constraint language. Therefore, we describe the literature findings for two DSL-based approaches. The authors of [49] support required and (mutual) excluded variabilities, i.e. a type of constraint language as discussed for basic variability modelling in Section 3.2. FAMILAR [1] supports propositional logic, i.e. is based on Boolean constraints discussed in Section 3.2.

Some benefits of using domain-specific concepts in general as well as in variability modelling are improved communication and understandability, increased expressiveness, a concise language, a typically higher level of abstraction, better scalability within the problem domain and an increased productivity [29, 32, 33]. In contrast, proper design of a DSL is a difficult task which implies upfront investments [32, 33]. Further, each DSL increases the existing zoo of languages aggravating the so called ghetto language problem, i.e. staffing becomes difficult due to the fact that in-house used DSLs are not known outside a company [29]. Moreover, combining DSLs into an amalgamated language becomes complicated when elements and concepts overlap and need specific approaches, as for example described in [14, 42].

In the remainder of this section, we will give examples for two basic strategies of using DSLs in variability modelling, namely defining a DSL for variability modelling in the context of our running example as well as extending a DSL by concepts from a general-purpose language. We will not give examples for the remaining strategies discussed above as we only want to provide some illustration using these examples.

```
1: ContentSharing –> ”Content-Sharing” App Platform
2: App -> ”Application” ContentType Container Database
3: ContentType -> ”Content-Type” (”Text” Video ”Audio” 3D BLOB)<1-5>
4: Video -> ”Video” “bitrate =” IntLiteral
5: …
6: IntLiteral -> -?(0..9)*
```

Figure 7: A DSL for configuring content sharing applications (fragment).

We illustrate the approach of defining a specific DSL for variability modelling in the context of the running example. In the sections above, the structure of the variability models, e.g., shown in Figure 6 was mainly determined by the decomposition hierarchy of configurable elements, i.e., a tree structure. We will rely on this tree structure to derive a textual concrete syntax for a domain-specific variability modelling language following the approach in [73]. For the presentation of the examples we use a notation inspired by the examples in [73]. Figure 7 depicts the grammar definition of a simple DSL for configuring content sharing applications. Similar to the variability models shown in the sections above, the grammar allows specifying the application part and the target platform (line 1). The application part (identified by the string “Application”) is subdivided into content type, container and database (line 2). Line 3 specifies that at least one and at maximum five content types may be selected for a concrete application, namely text, video (which is further

specified by the bitrate in terms of an integer literal), audio, 3D and BLOB. The translation of the remaining structure of the variability model to the DSL is straightforward and not detailed here. Two example configurations which differ in the selected content types are shown in Figure 8.

```
Content-Sharing                    Content-Sharing
  Application                        Application
    Content-Type                       Content-Type
      Text                               Text
      Video bitrate = 128                Audio
…                                  …
```

Figure 8: Example instantiations of the DSL in Figure 7 (fragment).

The DSL depicted in Figure 7 captures only the decompositions of the content sharing variability model shown in Figure 6, i.e. it allows specifying all possible configurations including those which are actually forbidden by constraints. To enable constraints in our example, we can extend the DSL by OCL [52] or Alloy constraints [38]. Further, we can extend the DSL by primitive data types e.g. taken from Java in order to avoid repeated definitions of well-known language constructs. Figure 9 illustrates the extended DSL and also shows one of the constraints used in the running example. By adding further language constructs taken from a programming language also Turing-complete constraints can be realized as for example done in DOPLER [24] by embedding Java fragments as constraints.

```
1: ContentSharing –> ”Content-Sharing” App Platform
2: App -> ”Application” ContentType Container Database
3: ContentType -> ”Content-Type” (”Text” Video ”Audio” 3D BLOB)<1-5>
4: Video -> ”Video” ”bitrate =” int
5:    [Tomcat implies Video.bitrate <= 128]
6: …
```

Figure 9: DSL extended by OCL constraints and primitive Java datatypes (fragment).

In this section, we have discussed the application of DSLs for variability modelling by identifying six different usage strategies from literature. On the one hand, defining own or extending existing variability modelling languages by required modelling concepts can greatly extend the expressiveness of a variability modelling approach. On the other hand, the analysability of variability models specified in DSLs suffers or, in extreme cases, is not possible anymore, e.g. when constraints given in a Turing-complete programming language are used. For the INDENICA variability modelling language we believe that DSLs may be considered for carefully extending a core variability modelling language, e.g. to realize several layers of expressiveness or to enable domain-specific constructs in order to simplify the application of variability modelling.

3.7 Summary

In the previous sections we discussed different categories of expressiveness of variability modelling concepts, ranging from less powerful to more powerful. The

discussion of each category included the introduced modelling elements and the dependency management capabilities. This provides a basis for our definition of the core variability modelling language in Section 5.

The above discussion of different modelling concepts together with our discussion of requirements in Section 2 showed that we will actually need a rather expressive modelling language. Thus, non-Boolean variability is a must from the point of view of industrial service-based systems. Configuration references as well as complex settings seem to be often relevant and are thus likely candidates for inclusion in the INDENICA variability modelling language. However, this must be done so that the base concepts do not become unnecessarily complex. In addition, a way for embedding DSLs must be found. Again, ease of use for the most standard issues is important, so we expect to rely on a simple core language which is extended in a way so that its use is not complicated for users who do not need the more advanced features.

4 Advanced Variability Modelling Concepts

The purpose of this section is to introduce additional variability modelling concepts that arise from the specific requirements regarding variability modelling in service-based systems. We discussed these requirements in Section 2.2. Thus, Section 4.1 will discuss required variability modelling capabilities in service ecosystems in general. In Section 4.2, we will discuss Quality of Service (QoS) and Service Level Agreements (SLA) and the variability modelling concepts used in literature in this context. Section 4.3 will discuss meta-variability. This describes the variability of aspects (e.g., binding time) that are often treated as constant in a variability implementation. Finally, Section 4.4 will discuss extensions to variability modelling that address on service-technology specific capabilities. We will discuss different service technologies and describe the mapping of relevant aspects to modelling elements.

4.1 Service Ecosystems

A service ecosystem is a large and non-trivial collection of different services that interact and cooperate across technology- and business-boundaries [60]. In the context of the INDENICA project we extend the notion of service ecosystems to service platform ecosystems in which multiple platforms (or parts of them) form an integrated and domain-specific platform. Key characteristics for variability modelling in service ecosystems can be transferred to service platform ecosystems. These are modularity and extensibility to support the integration of multiple variability models, their individual configuration as well as the configuration across variability model boundaries, and partial configuration to enable pre-configuration of modelling elements.

In this section, we discuss concepts for extending the basic INDENICA variability modelling language to satisfy the above characteristics and requirements of service (platform) ecosystems. In literature, different approaches are described that deal with composition and extensibility in variability modelling. Some of these approaches focus explicitly on the integration of models or modelling elements of different languages [27, 37]. These approaches are reasonable in the general context of software ecosystems but in the INDENICA project, we focus on the use of a single variability modelling approach. In the following, we will discuss specific modelling concepts that focus on the composition and extensibility of homogeneous models and modelling elements used in literature and conclude on relevant concepts for variability modelling in service (platform) ecosystems.

Czarnecki et al. [20] propose a cardinality-based feature modelling approach in which special leaf nodes act as extension points to connect additional feature models of the same approach using feature diagram references. This extension mechanism enables both the definition of constraints among features of the basic and the connected models and the (valid) configuration across model boundaries. The authors also introduce the notion of staged configuration meaning that certain stakeholders may eliminate only some configuration choices of a feature model

before (on the next stage) other stakeholders may configure the remaining variabilities. This requires a specialization process that transforms a feature model into another model, such that the latter model is a (specialized) subset of the former one. Rosenmüller et al. [61] integrate feature models of different Software Product Lines (SPLs) using composition models. This approach is based on object-oriented concepts in which a class represents a single SPL while an object of a class represents an instance of this SPL (the feature model and the model configuration respectively). This allows for the definition of constraints among multiple feature models on the level of composition model classes and instances. The latter defines restrictions similar to configuration references introduced in Section 3.5. The information of such a composition model serves as a basis for the automatic derivation of a configuration interface that guides a user through the configuration of all SPLs. El-Sharkawy et al. [26] use a variability modelling language based on [63] to model both, each SPL separately and the composed SPL. The common language enables simple composition of the homogeneous models. Namespaces ensure the clear identification of variabilities in the composed model. This facilitates the definition of constraints among modelling elements of the individual models.

In practice, additional problems and challenges of variability modelling in ecosystems arise. Brummermann et al. [13] discuss the challenges of distributed evolution of variability in information system ecosystems in the context of HIS GmbH. In this scenario third-party vendors and customers may add configurations or override (parts of) the base configuration that yield an unmanageable set of variabilities for the company. The results are version- and update-conflicts when the base system evolves. The authors address these conflicts by a formalism that supports structured difference calculations to identify the effective changes that occurred in a new version. Schmid [62] focuses on distributed software development and identifies characteristics of a variability management approach particularly suited for such development scenarios. He identifies a set of concepts introduced in Debian Linux [2] and Eclipse package management [70] that are helpful for distributed variability management: decomposition (assignment of responsibilities for different parts of a variability model to different teams), version-based dependency (usage of version information of the different parts to define what combinations are acceptable), information hiding (explicit definition of the visibility of variability), variability interfaces (extension points and their parameterization for extending base variability models) and inversion of dependency (additional variability models or modelling elements know the basic variability model that they extend, but not vice versa).

Based on the approaches for variability modelling in ecosystem we can sketch the realization of the following specific requirements listed in Section 2.2:

- Composition of multiple variability models, open variation and modularity (S12, S15, S16): Composition of variability models will be facilitated by extension mechanism like the definition of extension points and variability interfaces. This mechanism will allow the clear separation of base and additional modelling elements (including the responsibility for the specific parts) and enables the independent configuration of variability models and the configuration across model boundaries.

- Definition of constraints among composed variability models and clear element identification (S13): We will use the concept of namespaces to clearly identify each element of a composed variability model. This will also facilitate the definition of constraints across variability model boundaries as we can address each modelling element explicitly.
- Support for partial configuration (S14): We will include a mechanism to allow both, partial configuration as well as the redefinition of such configuration. The underlying concept will in particular support staged configuration. In this concept, (partial) configurations will be inherited from a basic model to provide a derived model, but remain reconfigurable to adapt the configuration to specific requirements.

4.2 Quality of Service and Service Level Agreements

The quality of service provisioning in service-based systems is typically regulated by Service Level Agreements (SLA). SLAs are formal specifications, which define the conditions under which a certain service is provided by a service provider to a service customer. The actual quality of service (QoS) may depend on various factors such as the current request rate, network usage or the overall utilization of the service hosting infrastructure. Maintaining a negotiated quality of service over time may lead to changes in service parameters, the service infrastructure configuration, the services themselves or the service composition.

In this section we discuss concepts for extending the basic INDENICA variability modelling language from the point of view of service quality support. Below, we discuss the specific modelling concepts used in literature in a) SLA modelling languages and b) in product line variability modelling. Finally, we conclude on relevant concepts for modelling quality in service variability models.

We compared the modelling concepts of nine specific SLA approaches, namely Language for defining Service Level Agreements (SLAng) [43], High-Level Objective-based Policy for Enterprises (HOPE) [57], Web Service Offering Language (WSOL) [71], WebService Agreement (WS-Ag) [50], Composite SLA management (COSMA) [45], Web SLA (WSLA) [41], Web Service Modelling Language (WSML) [68], QoS Modeling Language (QML) [30], UML QoS profile (UM2QoS) [15] as well as the current UML modelling extension for modelling and analysis of real-time embedded systems (MARTE) [54]. With respect to the requirements given in Section 2.2, we identified the following relevant concepts:

- **Typed attributes** are commonly used for expressing quality characterizations in terms of qualitative statements (e.g. based on enumerations) or quantitative (numeric) values. These attributes can be grouped using a structured data type such as a record as e.g. done in SLAng [43] or MARTE [54].
- Arithmetic expressions are used to express **derived quality characteristics**, such as quality functions in HOPE [57] or combined metrics in WSLA [41]. The Object Constraint Language (OCL) [52] is used in UM2QoS [15] and in MARTE [54].

- **Logic expressions** are used to specify constraints and restrictions on quality attributes. Various types of logic are used in literature, such as first-order predicate logic on quality attributes and derived quality characteristics in COSMA [45], typed predicate logic in WSML [68] or UM2QoS [15] (using OCL [52]), or even temporal logic as suggested in WS-Ag [50].

Quality specifications have been combined with variability modelling, e.g. to analyze variability models and to determine the overall quality of a derived product. By analyzing related publications, we identified the following concepts as being relevant for the requirements derived in Section 2.2:

- **Quantitative and qualitative attributes** are a general requirement for the integration of quality constraints into variability modelling [5]. Particularly, qualitative real values are used to characterize the footprint of the product in [66], reliability and energy consumption in [31] or cost and response time in [48]. Instead of attributes, model annotations are used in [31].
- A further requirement for quality specifications in variability modelling is the support of **algorithms for calculating derived values** [5]. One example of such general-purpose algorithms are the analysis functions in [31] while the authors in [48] rely on three specific aggregation operators, namely summation, multiplication and min-max calculation.
- Accordingly, the **constraints language** can be enabled for quality attributes, such as the probabilistic logic language in [31].

As an additional topic it might be beneficial to consider integration with **goal-oriented modelling** (as applied in the INDENICA requirements approach in WP1) in order to support traceability of requirements to the variability model. Such an integration is discussed in [22]. There the authors use positive and negative contributions to goals to derive quality properties of the variability model.

Based on the approaches for QoS specifications in SLA modelling and in variability modelling we can sketch the realization of the following specific requirements listed in Section 2.2:

- Specification of qualitative and quantitative QoS properties (S6): Qualitative and quantitative QoS properties will be expressed as non-Boolean attributes, i.e. qualitative properties as ordered enumerations and quantitative properties by numeric (real) values.
- Constraints on QoS properties (S8): We expect that typed predicate logic using OCL (more detailed BasicOCL or EssentialOCL) will be adequate for QoS constraints in INDENICA fostering the use of well-known constructs and standards. In contrast, probabilistic logic or even temporal logic would seriously impact the analyzability of variability models.
- Arithmetic expressions for derived quality properties (S7): Will be realized by defined operations in OCL (as selected for realization of S8).

4.3 Meta-Variability

Meta-variability is the capability of systematically varying an aspect of variability that is taken as a fixed part in a variability description. Examples include:

- Variability of variability description (e.g. modifying alternatives into options or adaptivity constraints)
- Changing binding times
- Changing variability implementation mechanisms

In this section, we first illustrate meta-variability in the running example and then describe extensions and concepts for supporting meta-variability. This provides a basis for the design of the corresponding capabilities in the INDENICA variability modelling language.

An important form of meta-variability is binding time variability. In the running example this leads to an augmentation of each variability by information about the latest point in time when the variability must be resolved. Let us assume that the domain engineer defines that compile time and runtime binding are both valid binding times for the `Content-Type` variability. When compile time is selected during product configuration, the set of supported content types as well as the related realization can be included into the product by the compiler. In contrast, when selecting runtime, all possible implementations of content type functionalities must be included at compile time. Further, additional functionality is required to switch among the different `Content-Type` implementations at runtime.

A meta-variability can be understood as an additional variable attribute assigned to a "main" variability. This additional attribute can by itself be defined, applied and constrained. Thus, the basic concepts of the INDENICA variability modelling language should be designed in a way that a natural extension to meta-variability is possible.

We will discuss the realization of the specific requirements defined in Section 2.2 below:

- **Definition of meta-variabilities and their properties (S9):** Before using a meta-variability in a concrete variability model, the meta-variability and its properties must be defined. This includes the concrete values a meta-variable can take as well as its cardinality specifying whether the meta-variability is an optionality, an alternative, a range, etc. As a meta-variability can be considered as a variability attribute, we just need to foresee structured type declarations indicating the new type as a meta-variability. As an example, the binding time range mentioned above can be defined as a meta-variability with values specified by an enumeration (compile time, deployment time, startup time, runtime) with cardinality 1..*, i.e. at least one binding time must be selected in a valid model.
- **Definition of the impact of meta-variabilities (S10):** The application of a meta-variability happens like the binding of variabilities by selecting concrete values for the attributes of variabilities during the configuration process. This

must respect the specific type definitions and properties defined for individual meta-variabilities.

- **Definition of constraints (S11):** The constraint language of the INDENICA variability model should be capable of defining constraints which refer to meta-variabilities so that bindings of variabilities and meta-variabilities can be constrained or automated. As an example, postponing the binding time of `Content-Type` to runtime should enforce that also container and database can be bound at runtime.

4.4 Service Technology-Specific Extensions

In deliverable D2.2.1, we discussed different variability implementation techniques from literature. These techniques are relevant to service-oriented development and, in particular, to the INDENICA project. We introduced a categorization of the techniques based on the variability objects they address (also shown in Section 2.2). While Deliverable 2.2.1 focused on variability implementation, in this section, we will discuss what can be learned from these techniques with respect to variability *modelling*. The first part of this section discusses the approaches that include variability modelling explicitly. In the second part, we consider the approaches that do not model variability explicitly and discuss why additional capabilities are needed.

The subset of approaches that use an explicit variability model use rather basic modelling capabilities (cf. Table 1). In these cases, the configuration space is typically described using typical configurable elements like features. All approaches use mandatory, optional and alternative forms of variations, while multiple selections and cardinalities are only used by few. Hierarchical structuring of configurable elements is inherent in the used modelling approaches (feature modelling or orthogonal variability modelling). The definition of constraints mostly relies on Boolean operators like requires and excludes.

An exception is the approach deployment / undeployment script, which additionally uses annotations to define deployment information for each configurable element [47]. The authors do not give any detailed information on how the deployment information is actually modelled, but we assume that this requires additional modelling elements like attributes and non-Boolean variability, e.g., strings to specify deployment locations. This approach may also use non-Boolean operators to restrict the deployment information defined in additional modelling elements. Again, this is an assumption due to the lack of information on how deployment information is actually modelled in this approach.

Table 1 summarizes the approaches to variability implementation with respect to the used variability modelling elements.

		Reflective Variability*	Event-based Composition*	Use Platform Management Services*	Component-based Service Implementation**	Class Wrapper**	Service Composition Generation***	Deployment / Undeployment Scripts****
Configuration Space Definition	Basic Elements	x	x	x	x	x	x	x
	Additional Elements							x
Forms of Variation	Mandatory	x	x	x	x	x	x	x
	Optional	x	x	x	x	x	x	x
	Alternative	x	x	x	x	x	x	x
	Multiple Selections	x	x		x	x		x
	Cardinality	x	x		x			x
Hierarchies		x	x	x	x	x	x	x
Constraints	Boolean	x	x	x	x	x	x	x
	Non-Boolean							x

*addresses variability in technical platform services
**addresses variability in domain-specific services
***addresses variability in service composition and processes
****addresses variability in service and platform deployment

Table 1: Summary of used modeling elements in variability implementation approaches

In deliverable D2.2.1 we identified approaches to variability implementation that do not explicitly model variability. An example is the component service replacement approach [44], which addresses variability in service composition and processes and relies on QoS calculation and the exchange of one or more services to meet QoS requirements at runtime. We discussed the modelling of QoS and SLA aspects in detail in Section 4.2. The scoping and fine-tuning approach provided by SAP[27] also addresses variability in service composition and processes. The approach provides a basic Business Adaptation Catalogue (BAC) and additional, predefined Business Configurations (BC). Customers use the BAC to select the required base functionalities and may overwrite parameters of the BC sets based on their specific needs. We discussed the issues of pre-configuration and overriding existing parameters, or more generally, overriding existing configurations in Section 4.1. The context-aware deployment plan approach [3] facilitates service and platform deployment variability. This approach does not use any specific modelling elements

[27] http://help.sap.com/saphelp_byd30/en/KTP/Software-Components/0120061532010000337 9/SAP_BBD/SAP_BBD.html

or approaches, but could be combined with one of the approaches introduced in Section 3.

The remaining variability implementation techniques allow arbitrary extensions to the base functionality of the respective variability objects. Existing variability modelling approaches are not well capable of describing this. This might be the reason, why those variability implementation approaches that address this do not explicitly use a variability modelling approach.

In summary, analyzing the capabilities of existing variability techniques for service implementation does not provide a good basis for solving the issues. Basic techniques are used, but more demanding aspects of variability modelling are typically not addressed, rather the implementations work without explicit models. In particular, the need to address arbitrary and unforeseeable extension to base functionalities is a major challenge, as we discussed above. We defined this as an essential requirement in the context of service and service platform ecosystems (S15) in Section 2.2. Addressing these challenging issues and integrating them with service platform technologies will be a major contribution of the INDENICA approach.

4.5 Summary

In the previous sections we discussed additional variability modelling concepts that arise from the specific requirements regarding variability modelling in service-based systems. The discussion of each concept included the introduction of modelling elements and dependency management capabilities. This provides a basis for our definition of the advanced variability modelling language in Section 5.

The above discussion of additional modelling concepts together with our discussion of requirements in Section 2 showed that we will actually need to extend the expressiveness of our core modelling language. Thus, composition of multiple variability models, constraints among composed models and support for partial configuration is a must from the point of view of service ecosystems. Meta-variabilities, the definition of their impact as well as the restriction of meta-variabilities seem to be often relevant and are thus likely candidates for inclusion in the INDENICA variability modelling language. Service technology-specific extensions could not be identified, however, the need to address arbitrary and unforeseeable extensions to base functionality is a challenging issue. This will be a major contribution of the INDENICA approach.

The specification of qualitative and quantitative QoS properties requires non-Boolean attributes as well as corresponding dependency management capabilities including arithmetic expressions to define restrictions on QoS properties. This is subsumed by the core variability modelling concepts that we introduced in Section 3.

5 The INDENICA Variability Modelling Approach

In this section, we will describe the concepts of the INDENICA Variability Modelling Language (IVML). In accordance to the previous sections, we will distinguish between a core modelling language and an advanced modelling language that extends the core language to satisfy the specific requirements that arise in the INDENICA project. This distinction facilitates ease of use for the most standard issues in variability modelling as it does not complicate the use of this language for users who do not need the more advanced features. The concepts of the core modelling language are based on the results of the discussion in Section 3. In this section, we discussed different levels of expressiveness for basic variability modelling in INDENICA. The core modelling language is extended by advanced modelling concepts that we identified as prerequisites to effective and efficient variability modelling in service-based systems and, in particular, in service (platform) ecosystems in Section 4.

The basic concepts of the IVML are related to approaches like the Text-based Variability Language (TVL) [12], the Class Feature Relationships (Clafer) [4], the Compositional Variability Management framework (CVM) [59], etc. However, we decided to develop a different approach, based on decision modelling concepts, in order to appropriately address the requirements identified in Section 2.4.

We will introduce a textual specification to describe the IVML concepts. This will help to give a precise representation of the modelling concepts. The syntax, we use in this section was developed as a basis for representing the concepts. The details of the syntax may change considerably in the course of the project. Our presentation of the IVML-syntax draws upon typical concepts used in programming languages, in particular Java, and other modelling languages such as TVL [12], Clafer [4], the Object Constraint Language (OCL) [52], or the UML [53]. The dependency management concepts of the IVML mostly rely on the concepts of the OCL. We will adapt these concepts as needed to provide additional operations required by IVML-specific modelling elements, e.g. match and substitute operations for decision variables of type string.

We will use the following styles and elements throughout this section to illustrate the concepts of the IVML:

- The syntax as well as the examples will be illustrated in `Courier New`.
- **`Keywords`** will be highlighted using bold font.
- *`Elements and expressions`* that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers of new types will start with a capital letter to easily distinguish them from variables.
- Expressions will be separated using semicolon “`;`”.

- Different types of brackets will be used to indicate lists "( )", sets "{}", etc. This is closely related to the Java programming language.
- We will indicate comments using "//" and "/* ... */" (cf. Java).

We will use the following structure to describe the different concepts:

- **Syntax:** this is the syntax of a concept. We will use this syntax to illustrate the valid definition of elements as well as their combination.
- **Description of syntax:** provides the description of the syntax and the associated semantics. We will describe each element, the semantics and their interaction with other elements in the model.
- **Example:** the concrete use of the abstract concepts is illustrated in a (simple) example.

In Section 5.1, we will describe the INDENICA variability modelling core language. We will introduce the required elements and expressions to define a basic configuration space including Boolean and non-Boolean variabilities. We will further describe the dependency management capabilities of this language to restrict configuration spaces. Finally, we will describe the definition of (product) configurations based on configuration spaces.

In Section 5.2 we will describe the advanced concepts of the INDENICA variability modelling language. We will introduce extensions that are required to satisfy the specific requirements in the INDENICA project like the support for service-ecosystems, for service technology and meta-variability.

Section 5.3 will provide a summary on how the requirements collected in Section 2 are implemented by the IVML, i.e. to which degree individual requirements are realized by the language.

5.1 INDENICA Variability Modelling Core Language

This section describes the core language of the IVML. In this language, a project is the top-level element that identifies the configuration space of a certain (software) project. In terms of a product line, this may either be an infrastructure as a basis for deriving products or a final product. In a project the relevant modelling elements will be defined. We describe this in the first part of this section. In the second part, we introduce the type system supported by the IVML. These types can be used to declare different types of decision variables. The dependency management capabilities to restrict the configuration space of a project will be described next. Finally, we will introduce the configuration concept of the IVML, which enables the definition of specific (product) configurations based on the configuration space defined in a project.

5.1.1 Projects

In the IVML a project (**`project`**) is the top-level element in each model. This element is mandatory as it identifies the configuration space of a certain software project and, thus, scopes all variabilities of that software project. The definition of a project requires a name, which simultaneously defines a namespace for all elements of this project.

Syntax:

```
project name {

  /* Definition of the configuration space and
  configurations. */

}
```

Description of syntax: the definition of a new project consists of the following elements:

- The keyword **`project`** defines that the identifier *name* is defined as a new project or, to be more precise, as a new configuration space.
- *name* is an identifier that defines the name of the new project and, thus, the namespace of all elements within this project.
- The elements surrounded by curly brackets define the configuration space of the new project.

Example:

```
project contentSharing {

  /* This will define a new project for a content-sharing
  project. This is related to our running example in
  Section 3. */

}
```

5.1.2 Types

In a project (cf. Section 5.1.1) different kinds of core modelling elements may be used to both represent the variabilities and define a configuration space appropriately. We will express these kinds as formal types in IVML, thus defining a (strongly) typed language. We distinguish between basic types, enumerations, container types, derived and restricted types and compound types. These types can be used to declare or define concrete decision variables.

5.1.2.1 Basic Types

In Section 3, we argued that non-Boolean variability is a must for the core expressiveness of the INDENICA language. Thus, the IVML supports as basic types Boolean (**`Boolean`**), integer (**`Integer`**), real (**`Real`**) and string (**`String`**) with their usual meaning. The names of the basic types are aligned to OCL [52]. These types support the definition of basic variabilities, e.g. the `Boolean` type may be used for modelling optional variabilities. In addition, types like `Integer` or `Real` provide a basis for defining advanced variabilities, e.g. using an `Integer` to define a quantitative property for Quality of Service (QoS) as described in Section 4.

5.1.2.2 Enumerations

Enumerations allow the definition of sets of named values. This is used to describe a set of possible resolutions of a decision.

Syntax:

enum *Name*$_1$ {*value*$_1$, ..., *value*$_n$};

enum *Name*$_2$ {*value*$_1$=*n*$_1$, ..., *value*$_n$=*n*$_n$};

Description of syntax: the definition of a new enumeration type consists of the following elements:

- The keyword **enum** defines that the identifier *Name* is defined as a new enumeration.
- *Name* is an identifier and defines the name of the new type.
- The identifiers surrounded by curly brackets are the concrete elements of the enumeration. A specific element of an enumeration can be accessed using the "."-notation, e.g. *Name*$_1$.*value*$_1$.
- Specifying concrete numeric values for elements of an enumeration (*value*$_i$=*n*$_i$) turns the enumeration into an ordered enumeration. This enables relations like greater than (>) or less than (<) and operations like next (**next**) or previous (**previous**) on the values to be used.

Example:

```
enum Colors {green, yellow, black, white};

enum BindingTimes {configuration=0, compile=1,
  runtime=2};
```

5.1.2.3 Container Types

The IVML provides two container types, sequences and sets. Sequences can contain an arbitrary number of elements of a given content type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. These types can be used to describe a number of possible options out of which several can be selected at the same time. Elements in a container (both sequences and sets) can be accessed by their position in the container using an index ([*index*]).

The IVML supports a set of operations specific for container types, e.g. adding or appending elements to a container, deleting elements of a container, selecting specific elements, etc. We will introduce the full set of operations in Section 5.1.4.

Syntax:

```
// Declaration of a new sequence and a new set.

sequenceOf(Type) Name1;
```

```
setOf(Type) Name2;


/* Access to elements of a variable of a container type:
this holds for both sequences and sets. We will discuss
variables in Section 5.1.3. */

Name1 variableName;

variableName[index] = value;
```

Description of Syntax: the definition of a container type consists of the following elements:

- The **sequenceOf** and **setOf** keywords indicate the definition of a new container of the respective type followed by the *Type* of the elements contained in brackets.
- The identifiers $Name_1$ and $Name_2$ are the names of the new containers.
- Accessing a specific element of a container type (variable) requires the specification of an index ([*index*]). An index is a positive integer value specifying the position of an element in a container. Accessing a specific position is only a valid operation, if this position has previously been set by different means like the **add** function (the set of operations is introduced in Section 5.1.4).

Example:

```
/* Definition of a new enumeration. "blob" means "binary
(large) objects". */

enum ContentType {text, video, audio, threeD, blob};


/* Denotes types of contents supported by a system */

sequenceOf(ContentType) Contents;

Contents basicContents = {text, audio}
```

5.1.2.4 Type Derivation and Restriction

The IVML allows the derivation of new types based on existing types. This supports extensibility and adaptability as users may define their own types based on basic types, enumerations or container types as well as on previously derived types. The derivation may also include restrictions to the existing type, e.g. to restrict the possible values of the new type to a subset of the values of the existing type. The restrictions are defined by one or more constraints (we will discuss constraints in detail below). Multiple constraints are implicitly combined by a Boolean OR. Thus, at

least one constraint has to be satisfied by the new type. The constraints will be defined in OCL style as described in Section 5.1.4.

Syntax:

```
typedef Name1 Type;

typedef Name2 Type with (constraint1, ...,
  constraintn);
```

Description of Syntax: the definition of a derived type consists of the following elements:

- The **typedef** keyword indicates the derivation of a new type based on an existing type.
- The identifiers *Name*$_1$ and *Name*$_2$ are the names of the new types.
- The identifier *Type* denotes the basic type from which the new type (*Name*$_1$or *Name*$_2$) will be derived.
- The optional keyword **with** introduces a non-empty set of constraints, surrounded by brackets, out of which at least one must hold for *Name*$_2$. In case of deriving *Name*$_2$ from **String** the constraints may define regular expressions.

Example:

```
/* Definition of a type "AllowedBitrates" which is a set
of Integers, i.e. a kind of alias for a complex type
definition. */

typedef AllowedBitrates setOf(Integer);


/* A new modelling type of the basic type integer that is
restricted to assume values between "128" and "256". */

typedef Bitrate Integer with (Bitrate >= 128 and
  Bitrate <= 256);
```

5.1.2.5 Compounds

A compound type groups multiple types into a single named unit (similar to structs or records in programming languages or groups in feature modelling). This allows combining semantically related decisions from which each element has to be configured individually.

Syntax:

```
compound Name {

  Type name1;
```

```
  ...
}
```

Description of Syntax: the definition of a compound type consists of the following elements:

- The **compound** keyword indicates the definition of a new compound type.
- The identifier *Name* defines the name of the new compound type.
- The set of elements surrounded by curly brackets defines the types of the compound type. Each declaration of a typed element is separated by a semicolon.

Example:

```
/* A new compound type for the configuration of different
(web) content. The content may vary in terms of name and
bitrate. "Content.bitrate" is the integer within the
compound content. */

compound Content {

  String name;

  Integer bitrate;

}
```

5.1.3 Decision Variables

The types introduced in Section 5.1.2 can be used to declare (decision) variables representing a concrete variability. A decision variable is an element of a project (configuration space) that basically accepts any value of its type. Constraints may further restrict the possible values by removing certain combinations of values from the allowed configuration space. The value given to a decision variable defines the variant of the represented variability.

In IVML a decision variable may either be declared with or without a default value (this is an optional parameter). Decision variables with a default value can be further configured by overwriting their (default) value at a later point in time. However, overwriting the default value is not necessary.

Syntax:

```
// Declaration of a decision variable.

Type name1;


/* Declaration of a decision variable with a default
value. The "valueAssignment"-expression will be described
in detail below. */
```

Type $name_2$ *= valueAssignment*;

Description of Syntax: the basic declaration of a new decision variable (excluding the declaration of an optional default value) consists of the desired type (one of the basic types, an enumeration, a container type, a derived or a restricted type, or a compound type) followed by an identifier ($name_1$) that states the name of the variable.

Optionally, a default value can be assigned to a decision variable appending "=" followed by a *"valueAssignment"*-expression after the name ($name_2$) of the decision variable. The form of the *"valueAssignment"*-expression depends on the specific type of the declared decision variable:

- Basic types and Enumerations: an expression that yields a value of the corresponding type and can be actually calculated, i.e., it either consists of constants or the values of the variables are known.
- Container types: either an expression of the type of the container, which can be statically evaluated, or a set of values separated by commas in curly brackets after the name of the decision variable. The allowed values within the curly brackets are determined based on the base type of the container.
- Compounds: either an expression of the type of the compound, which can be statically evaluated, or a set of individual assignments, given in curly brackets. Each assignment explicitly gives the field in the compound that the assignment is made to, followed by a "=" and an expression of the corresponding element type. Again this expression needs to be statically evaluated.
- Derived type: the assignment follows the rules of the base type.

Example:

```
/* Declaration of a new variable of type integer with a
default value. */

Integer bitrate = 128;


/* Declaration of a new variable of type enumeration with
a default value (cf. Section 5.1.2.2). */

Colors backgroundColor = black;


/* Declaration of a new variable of type container
(sequence) with default values (cf. Section 5.1.2.3). */
```

```
Contents baseContent = {text, audio};


/* Declaration of a new variable of type compound with
default values (cf. Section 5.1.2.5). */

Content complexContent = {name = "Text",
  bitrate = 128};
```

5.1.4 Constraints

Constraints are used to define validity rules for a variability model, e.g. by specifying dependencies among decision variables. The syntax of constraints in the IVML basically follows the structure of expressions in propositional logic and, thus, is composed of:

- Simple sentences, which represent constants, decision variables and types which can be named by (qualified) identifiers.
- Compound sentences created by applying the operations to simple sentences and, in turn, to compound sentences. A correct compound sentence requires that the arguments passed to operations match the arity of the operation and the types of the parameters or operations comply, respectively.

The operations available in IVML as well as the type compliance rules will be discussed in the remainder of this section.

The constraints in IVML will mostly rely on the relevant part of the syntax as well as on a large subset of the operations defined in OCL. Table 2 summarizes the operations that are part of the IVML. In IVML we will use the constraint expression syntax of OCL, but omit the OCL contexts used to relate constraints to UML modelling elements. Two examples for such constraints are given below, one propositional and one first-order logic example using a quantifier:

```
(10 < a and a < 20) implies b = a
```

If a is in the range (10; 20) this implies that b has the same value as a.

```
mySet->forAll(x|x > 100)
```

All elements in `mySet` must be larger than 100

Regarding collections, we will take over the OCL collections `Set` and `Sequence`, and exclude `OrderedSet` and `Bag` in the initial versions of IVML. The class of collection operations used to construct iterator expressions (`select`, `reject`, `collect`) and, in particular, quantors known from propositional logic (`exists`, `forAll`, `isUnique`, etc.) impose specific challenges to the analysability of variability models, but are required for variability modelling in complex realistic settings.

In addition to the OCL operations listed in the bottom part of Table 2, the IVML extends this by the following operations: regular expressions on strings and operations for sets and lists. As syntactic sugar we will we provide index-based access to a `List` using the usual array notation. Further, we will introduce a generic

functionality for defining aggregation operations on entire lists: In addition to the OCL operation collect, which applies a function to all elements in a set, we will introduce the `apply` operation, which aggregates the result over a set using a binary, commutative function and a neutral element.

We will also adhere to the (usual) operator precedence rules defined in the OCL specification in Section 7.4.7 and 9.3.2, such as multiplication and division having precedence over addition and subtraction.

Type conformance will be defined by the following set of rules inspired by OCL (cf. OCL section 7.4.5):

- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real.
- Enums, sets and lists are not compliant among each other.
- The application of the **`refines`** keyword induces a hierarchy of compounds where the subtypes are compliant to their parent types, i.e. the parent type may be replaced by each subtype.
- Derived types are compliant to their base type as long as no constraints are specified.

The notation as well as the semantics of constraints in the IVML will be closely aligned to the OCL. As we will not take over all elements of the OCL, particularly not those specifying the application of constraints in the context of UML models, we will reuse selected parts of the OCL syntax and semantics where applicable. The top part of Table 2 summarizes relevant parts of the OCL serving as a basis for the syntax and semantics of constraints in the INDENICA modelling language.

Topic	Contribution to IVML	OCL Section
Basic types	Boolean, Integer, Real, String	7.4
enums	Definition of enums*, element access will be denoted by '.' instead of '#'*	7.4.2
let-expressions	as defined in OCL	7.4.3
additional operations	definition of named arithmetic expressions using the **def** keyword	7.4.4
Attribute access	as defined in OCL	7.5.1
Pathnames	as defined in OCL	7.5.7
Tuples	*similar to compounds in IVML, also defining the configuration of a compound*	7.5.15
Set operations	as defined in OCL	7.6.1-7.6.5
Concrete syntax	*selected parts, where applicable*	9
Constraint semantics	*selected parts, where applicable*	10
Formal semantics	*selected parts, where applicable*	Annex A
Operations		
All types	=, <>, != *addition: == as an alias for* =	11.5.2
Boolean	not, or, xor, and, implies	11.5.4
Real	-[28], abs, floor, round, +,-,*, /, min, max, <, <=, >, >=	11.5.1
Integer	-[9], +,-,*, /, abs, div, mod, min, max, <, <=, >, >=	11.5.2
String	size, toInteger, toReal, concat, substring; *addition for regular expressions:* matches, substitutes	11.5.3
Set, Sequence	size, includes, excludes, count, isEmpty, sum, product + min, max, avg, exists, forAll, isUnique, any, one, collect, select, reject, asSet, asSequence, *addition:* apply	11.6.1, 11.7.1, 11.9.1
Set	union, =, intersection, including, excluding	11.6.2, 11.7.2, 11.9.2
Sequence	union, append, prepend, insertAt, subSequence, at, indexOf, first, last, *addition:* []	11.6.5, 11.7.5, 11.9.4

Table 2: OCL parts taken over into IVML, changes and additions are given in italics.

5.1.5 Configurations

The IVML does not differentiate between a configuration space and specific (product) configurations. Instead, a project can simultaneously describe or extend a configuration space and define a configuration. However, typically a project will provide a configuration space, while a different project may extend it, while providing configurations information for the initially specified configuration space. The set of decision variables and constraints of a project represent the set of all

[28] Unary negation operator.

possible configurations. In addition, default values of decision variables as described in Section 5.1.3 define basic configurations and, thus, do not need to be further configured, but can be overwritten later as well. In addition, some values of decision variables can be derived using constraints. Any configuration, independent of where the values come from, must comply with the relevant constraints.

Configurations in the IVML do not require any specific or additional keyword. They are simply given by variable assignments. We illustrate this concept by a simple example.

Example:

```
/* A project that represents both a configuration space
and a configuration. The constraint implies a valid
configuration with a bitrate value between "128" and "256"
and "content = text" (if no further configuration is
done). */

project contentSharing {

  enum ContentType {text, video, audio, threeD, blob};

  typedef Bitrate Integer with (Bitrate >= 128 and
    Bitrate <= 256);

  ContentType content;

  Bitrate contentBitrate = 128;

  contentBitrate = 128 implies
    content = text;

}
```

5.2 Advanced Concepts of the INDENICA Variability Modelling Language

This section describes advanced concepts of the IVML. We will describe how to assign additional attributes to modelling elements. This allows describing certain modelling elements in more detail, e.g. assigning meta-variability information as described in Section 4. We then augment the compound types introduced in Section 5.1.2.5 by extension and referencing concepts. Extension concepts will also be introduced for projects (cf. Section 5.1.1), which cover modularization aspects as well as facilitating project composition. We will describe advanced configuration concepts including partial configurations as well as “freezing” configurations. Finally, we will describe a lightweight concept for including DSLs as part of a variability model.

5.2.1 Attributes

In the IVML modelling elements can be attributed by further (orthogonal) configuration capabilities, e.g. to express meta-variability such as binding times. An

attribute in IVML is basically a decision variable that is attached to another modelling element describing this element in more detail. Thus, an attribute may also have a default value and may be restricted by constraints (cf. Section 5.1.4). The impact of an attribute depends on the element it is attached to. In the IVML the following modelling elements can be attributed:

- Decision variable: attributes that are attached to a decision variable only describe this variable further. Depending on the type of the decision variable, the attributes of the variable also describe its elements, e.g. the various fileds of a compound variable. These fields may have additional attributes. Changing the value of a decision variable attribute will not cause any modification to elements outside the scope of the specific variable (as far as they are not connected by constraints).
- Project: attributes that are attached to a project will affect all variables of this project.

As the different elements may be nested, different values can be given for the same attribute on the outer and the inner scope.

Syntax:

attribute *Type* $name_1$ **to** $name_2$;

attribute *Type* $name_3$ = *value* **to** $name_4$;

Description of Syntax: the definition of an attribute consists of the following elements:

- The **attribute** keyword indicates the definition of a new attribute.
- The expressions *Type* $name_1$ and *Type* $name_3$ correspond to the definition of a decision variable described in Section 5.1.3 while $name_1$ and $name_3$ are the identifiers of the new attributes.
- The **to** keyword indicates the attachment of the new attribute on the left side to the element ($name_4$) denoted on the right side.
- $name_4$ may be one of the elements described above to which the attribute is attached.
- Optionally, a default value (*value*) can be assigned to the attribute by appending a value expression after $name_3$.

Example:

```
project contentSharing {

  enum BindingTimes {configuration=0, compile=1,
     runtime=2};

  // Attaching an attribute to the entire project.
```

```
  attribute BindingTime binding = compile
    to contentSharing;

}
```

5.2.2 Advanced Compound Modelling

In Section 5.1.2.5 we introduced the compound types to group multiple types into a single named unit. In this section, we will extend the modelling of compound types by refinement and referencing concepts. Refinement allows extending existing compound types by additional elements, yielding a new (extended) compound type. Referencing enables the definition of references to other elements like other compounds.

5.2.2.1 Extending Compounds

In the IVML a compound may extend the definition of a previously defined (parent) compound. This is indicated by the **refines** keyword. Extending compound types is similar to subclassing in object-oriented languages, i.e. *parentType* becomes a subtype of *compoundType* and *compoundType* may define further decision variables.

Syntax:

```
compound Name1 refines Name2 {

  // Define additional elements.
}
```

Description of Syntax: the definition of an extended compound type consists of the following elements:

- The **compound** keyword indicates the definition of a new compound type.
- The identifier $Name_1$ defines the name of the new compound type.
- The **refines** keyword indicates that the new compound type ($Name_1$) is an extension of a previously defined compound type ($Name_2$).
- The set of elements surrounded by curly brackets defines the additional elements that make up the extensions to the inherited elements of compound $Name_2$.

Example:

```
/* A compound type for the configuration of different
(web) content. */

compound Content {

  String name;

  Integer bitrate;
```

```
}

/* A new compound type that refines the previous compound
type. "ExternalContent" will subsume all elements of
"Content" and all additional elements defined below. */
compound ExternalContent refines Content{
  String contentPath;
  String accessPassword;
}
```

5.2.2.2 Referencing Elements

The IVML supports referencing of (other) elements, for example, other compounds within a compound type. A reference allows the definition of individual configurations of an (external) element for the referencing element without including the external element as part of the referencing element explicitly. This is indicated by the **refto** keyword used for the definition of a reference and the **refby** keyword that indicates the configuration of a referenced element.

Syntax:

```
project name1 {
  compound Name2 {
    Type name3;
    ...
  }

  // Declaration of a new reference.
  refto(Name2) Name4;

  // Configuration of a referenced element.
  refby(Name4).name3 = value;
}
```

Description of Syntax: the definition and the configuration of a reference consist of the following elements:

- The **refto** keyword indicates the definition of a new reference.
- *Name*$_2$ defines the referenced element (type).

- *Name*$_4$ is an identifier and defines the name of the new reference. In the IVML a reference is type, thus, the identifier for a new reference starts with a capital letter.
- The **refby** keyword indicates the configuration of a reference (the configuration of the referenced element respectively).
- *Name*$_4$ is an identifier that defines the reference to be configured.
- The syntax for configuring a reference depends on the type of the referenced element (see Section 5.1.3 for the syntax for assigning values to variables of a specific type). In the case above, we use "**.**"-notation to configure a single element of a referenced compound type.

Example:

```
/* A compound type for the configuration of different web
containers being responsible for serving web content. */

compound Container {

  String name;

  ...

}


/* Another compound type for the configuration of
different (web) content referencing the "Container" type
to configure its individual web container. */

compound Content {

  String name;

  Integer bitrate;


  // Declaration of a reference to the Container compound.

  refto(Container) myContainer;


  // Configuration of the above reference.

  refby(myContainer).name = "ContentContainer";

}
```

5.2.3 Advanced Project Modelling

In Section 5.1.1, we introduced the concept of projects (**`project`**) as the top-level element in each IVML-model. In this section, we extend the modelling capabilities of the IVML regarding projects in three ways: first, we describe versioning of projects that enables the definition of the current state of evolution of a project. This concepts correlates with the second concept: project composition. This introduces the capability of deriving new projects based on definitions in other projects and explicitly excluding certain projects from the composition. As part of this version information can be used. The third concept is project interface. The concepts of project composition and project interfaces support effective modularization and reuse of projects and, thus, configuration spaces.

5.2.3.1 Project Versioning

In IVML, projects can be versioned to define the current state of evolution of a project (and the represented product line infrastructure). Evolution of software may yield updates to projects. This can be described by a version. For defining a version, the **`version`** keyword is followed by a version number. This must be the very first element of the respective project. The version number consists of integer values separated by "`.`" assuming that the first value defines the major version, while following numbers indicate minor versions. The level of detail of version numbers is determined by the domain engineer.

Syntax:

```
project name {
  // Definition of a version for this project
  version Number.Number;
  ...
}
```

Description of Syntax: the attachment of a version to a project consists of the following elements:

- The **`version`** keyword indicates the definition of a new version for the project *`name`*.
- *`Number.Number`* defines the actual version of the project (here only two parts).

Example:

```
project contentSharing {
  version 1.0;
  ...
}
```

5.2.3.2 Project Composition

The IVML supports the composition of different projects. This is closely related to multi software product lines [61] and product populations [72]. Project composition allows to effectively reusing existing projects by using these projects within other projects. This also supports the decomposition of large variability models as semantically related parts can be defined in individual projects. The complete project then uses these (sub-) projects to define the combined project. In the IVML the following keywords are introduced for project composition:

- **import**: this keyword indicates the use of a project. This keyword allows using certain elements of a project by reference. If a project contains explicit interfaces (see below), the specific interface, which is used, must be given.
- **conflicts**: this keyword indicates incompatibility among projects. All projects (names) followed by this keyword cannot be used in combination with the project that defines this conflict expression. This is also checked for indirectly used projects.

The keywords **import** and **conflicts,** introduced above, can be combined with version expressions using the **with** keyword and the version-information of a project introduced in Section 5.2.3.1.

Syntax:

```
project name1 {

  /* This introduces the project name2. Optionally, a
  version may restrict name2 to a specific version as it
  is shown below. */

  import name2;

  // Accessing elements of a project.

  name2::element;


  /* This introduces incompatibility of project name1 with
  project name3 of version greater than Number.Number. */

  conflicts name3 with (name3.version > Number.Number);

}
```

Description of syntax: the definition of a new project composition consists of the following elements:

- The keyword **import** indicates that the entities, which are made available by the project or interface *name₂* will be available within the current project.

- For disambiguation the elements of *name*$_2$ can be accessed using the "::"-notation to express qualified names. If there is no ambiguity, they can be used directly.
- The keyword **conflicts** indicates incompatibility of project *name*$_1$ with project *name*$_3$.
- Optionally, version-expressions can be combined with the keywords **import** and **conflicts** using the **with** keyword. This defines specific versions of other projects to be imported into the current project or conflicting with the current project.
- A version expression includes the version-information of a project (cf. Section 5.2.3.1), a relation operator and a version number or a version-information of another project. In addition, logical operators can be used to concatenate simple version-expressions to define ranges of versions.

Example:

```
project application {

  /* This will define a new project for content-sharing
  applications. */

  String name;

}


project targetPlatform {

  // This will define a new project for target platforms.

  version 1.5;

  String name;

}


project contentSharing {

  /* This will define a new project for a content-sharing
  project importing two sub-projects "application" and
  "targetPlatform". The latter sub-project must be of
  version "1.3" or higher. */

  import application;

  import targetPlatform
    with targetPlatform.version >= 1.3;
```

```
    // Accessing the elements of the sub-projects.

    application::name = "myApp";

    targetPlatform::name = "myPlatform";

}
```

5.2.3.3 Project Interfaces

By default, all elements defined in a project are visible when they are imported into another project. In order to support effective modularization and reuse of variability models, we introduce interfaces to projects. Interfaces reduce the complexity in large-scale projects and provide means to automate the configuration of lower-level decisions based on high-level decisions.

Interfaces in a project define all elements of a project, not part of the interface, as private and, thus, make them invisible to the outside. This is indicated by the **`interface`** keyword within a project. In order to access any elements they need to be declared as parameters of the interface. This can be done by exporting existing variables (using the **`export`** keyword) or by declaring new parameter variables. As a special characteristic of the IVML, it is also possible to define multiple interfaces for the same project. This is different from other variability modelling languages like the CVL [55].

Importing a project (cf. Section 5.2.3.2) that includes interfaces allows the importing project to access only the parameters defined in the interface. All other elements of the project are not visible to the importing project.

Syntax:

```
project name1 {

    /* Declaration of a (private) decision variable. This
    variable is exported by the interface Name2. */

    Type name3;


    // Definition of a new interface.

    interface Name2 {

        /* Denotes the export of an existing decision variable
        of the project name1. */

        export name3;

        ...

    }
```

```
}
```

Description of syntax: the definition of a new project interface consists of the following elements:

- The keyword **`interface`** indicates the definition of a new interface of the project $name_1$.
- The keyword **`export`** indicates the export of the following decision variable $name_3$.

Example:

```
project application {
  // Declaration of (private) decision variables.
  String name;
  String appType;
  Integer bitrate;

  // Definition of a constraint.
  appType = "Video" implies bitrate = 256;

  // This will define an interface for this project.
  interface MyInterface {
      export name, appType;
  }
}

project contentSharing {
  /* This will import the interface "MyInterface" of
  project "application". */
  import application::MyInterface;

  /* Only the parameters of the interfaces are accessible.
  "application::bitrate" yields an error. As long as the
```

```
    variable names are unambiguous, the fully qualified must
    not be used. */

    name = "myApp";

    appType = "Video";

}
```

5.2.4 Advanced Configuration

In Section 5.1.5, we introduced the configuration concept of the IVML. In this section, we will extend this concept to partial configuration. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring only parts of the project. The set of all configuration steps typically yield a full configuration of the entire project. We will further introduce the concept of persistent (parts of) configurations. We call this “freezing”. Freezing (parts of) configurations defines these parts to be persistent. Persistent parts cannot be changed anymore in further configuration steps. Finally, we will describe how (parts of) configurations can be evaluated independently from other parts of the configuration. This allows deriving additional configuration values based on existing configurations using the constraints and value propagation.

5.2.4.1 Partial Configurations

The IVML supports partial configurations. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring only parts of the project. The set of all configuration steps typically yields a full configuration of the entire project. The configuration of a part of a project may also be reconfigured by the next configuration step (cf. the concept of default values, which we introduced in Section 5.1.3). For example, a service provider may define a (pre-) configuration of the provided service, while a service consumer may reconfigure his service to satisfy his specific needs.

Partial configuration in the IVML is a straight-forward consequence of the concepts introduced so far. We illustrate this concept by a simple example.

Example:

```
project application {

    /* This defines a new project for content-sharing
    applications including the (pre-) configuration of the
    configuration element. This is also the first
    configuration step.*/

    String name = "Application";

}


project targetPlatform {
```

```
    /* This defines a new project for target platforms
    without any configuration. */

    String name;

}


project contentSharing {

    /* This defines a new project for a content-sharing
    project and imports two sub-projects "application" and
    "targetPlatform". */

    import application;

    import targetPlatform;


    /* This is the second configuration step, including the
    re-configuration of the name-element of the sub-project
    "application" and a configuration of the name-element of
    the sub-project "targetPlatform". */

    application::name = "myApp";

    targetPlatform::name = "myPlatform";

}
```

5.2.4.2 Freezing Configurations

In the previous section we described the concept of partial configuration. This included the possibility to re-configure existing (pre-) configurations. Although re-configuration is reasonable in some cases, e.g. to modify a given configuration to satisfy an individual need, at the end we desire a persistent configuration to define a specific product. For example, service consumers should not be able to reconfigure some parts of a configuration defined by a service provider.

We introduce the concept of “freezing” configurations. This is indicated by the keyword **`freeze`**. Freezing configurations define the current (partial) configuration to be persistent. Persistent configurations cannot be changed anymore in the course of the configuration. Excluding elements of a configuration from being frozen, e.g. freezing only some elements of imported projects or a compound type, the **`but`** keyword can be attached after a freeze-expression. All elements followed by a but-expression will not be frozen.

Syntax:

```
project name1 {

    // Definition of new compound type
```

```
    compound Name2 {
      Type name3;
      Type name4;
    }

    /* Declaration of a new decision variable of the above
    type */
    Name2 name6;

    /* Freezing the configuration of the decision variable
    except element name4. */
    name6.name3 = value1;
    freeze {
      name6;
    } but (name6.name4)
  }
```

Description of syntax: the definition of persistent (parts of) configurations consists of the following elements:

- The keyword **freeze** indicates that all elements with their current values within the following curly brackets are persistent.
- Optionally, the keyword **but** indicates a set of elements that is excluded from being persistent. All elements of this set can be further configured. The but-expression may also include wildcards (*) which are necessary especially in large models. Attaching a wildcard to an element, e.g. *name6.**, yields all elements of *name6* to be excluded from being frozen.

Example:

```
  project application {
    /* Definition of a new compound type for the
    configuration of the content type of an application. */
    compound ContentType {
      String contentName;
      Integer bitrate;
```

```
  }

  // Declaration of a decision variable of the above type.

  ContentType appContent;

  /* Definition of the content name to be persistent. The
  required bitrate for this content may be configured as
  part of the configuration of the container type for this
  content. */

  appContent.contentName = "Text";

  freeze {

    appContent;

  } but (appContent.bitrate)

}
```

5.2.4.3 Partial Evaluation

The IVML provides a concept for the evaluation of configurations. This is indicated by the keyword **eval**. At the end of a project definition an implicit **eval** occurs. The explicit invocation of **eval** can be used to structure the definition of the variables and thus reduces the search-space during constraint-evaluation. **eval**-blocks are evaluated inside-out before the project itself is evaluated. **eval**-blocks on the same nesting level do not imply any evaluation sequence.

Syntax:

```
/* Evaluate a constraint that defines the relation between
two variables of the same type. This leads to the
assignment of the variable values to the unassigned
variable upon exit of the scope of the eval-statement. */

eval {

  name₁ = name₂;

}
```

Description of syntax: the evaluation of a configuration requires an **eval**-statement using the keyword **eval** followed by curly brackets.

Example:

```
project application {

  /* Definition of a new compound type for the
  configuration of the content type of an application. */

  compound ContentType {
```

```
    String contentName;

    Integer bitrate;

  }

  // Declaration of a decision variable of the above type.

  ContentType appContent;

  /* Definition of the content name and bitrate. This
  configuration is evaluated explicitly to minimize the
  search space. */

  eval {

    appContent.contentName = "Text" implies
      appContent.bitrate = 128;
  }

}


project targetPlatform {

  /* Define a new project for target platforms without any
  configuration.*/

  String name;

  Integer bitrate;

}


project contentSharing {

  /* Define a new project for a content-sharing project
  importing two sub-projects "application" and
  "targetPlatform".*/

  import application;

  import targetPlatform;

  /* This constraint restricts the bitrate of the target
  platform to be equal or greater than the bitrate of the
  application content. The bitrate of the target platform
  can be derived from the bitrate of the application
  content: "targetPlatform::bitrate = 128". At the end of
  a project definition an implicit evaluation for the
  whole project is done. */
```

```
    targetPlatform::bitrate
      >= application::appContent.bitrate;

}
```

5.2.5 Including DSLs

The IVML includes a lightweight concept for including domain-specific languages (DSLs) as part of the variability model. This supports situations, in which the variability may be expressed more intuitively or more naturally using DSLs.

DSLs can be embedded in IVML in terms of external language sections similar to inline assembler code in higher languages. The embedded DSL code is preprocessed in order to consider actual decision values during DSL evaluation, passed to a DSL-specific tool for evaluation and the result of the evaluation is considered as part of the actual IVML model, which triggered the evaluation. The evaluation result is interpreted as a part of the final IVML description.

Syntax:

```
DSL(stopString, prefix, dslInterpreter)

  // here goes the DSL

DSL(stopString)
```

Description of syntax: an external language section for a DSL is introduced by the keyword **DSL**. The parameters of the opening **DSL** keyword are:

- The *stopString* identifier is a string used for uniquely identifying the end of the DSL in combination with the **DSL** keyword. The part between the opening **DSL** keyword (excluding its parameters in parentheses) and the closing **DSL** keyword (marked by the *stopString*) is not analyzed by the IVML tools but passed to an external DSL interpreter for evaluation.
- The *prefix* identifier is a string identifying a DSL-specific prefix for IVML identifiers denoting decision variables. When passing the DSL code to the DSL specific tools, all occurrences of decision variables marked by the *prefix* are replaced by actual values for the individual decisions.
- The *dslInterpreter* identifier is a string containing, for example, a file name or an URI specifying the concrete DSL tool which is responsible for evaluating the instantiated DSL code, i.e. after substituting occurrences of decision variables.

Example:

```
project application {

  /* Declaration of a decision variable with a default
  value. */
```

```
    Integer bitrate = 128;


    /* Declaration of an embedded DSL section within an IVML
    project. */

    DSL("dsl.com","$","http://www.dsl.com/dslInterpreter")

      /* The actual DSL statements will be placed between
      the DSL keywords. */

      ...

      /* Applying IVML decision variables to DSL statements
      by using the DSL-specific prefix "$" defined above. */

      ... $bitrate ...

    DSL("dsl.com")

  }
```

5.3 Fulfillment of Requirements

In this section we will provide a summary on how the requirements collected in Section 2 are implemented by the IVML, i.e. to which degree individual requirements are realized by the language.

General Variability Modelling Requirements

- **G1**: Definition of a configuration space that represents all variabilities as a composition of basic modelling elements. The IVML realizes G1 by decision variables (Section 5.1.3) of the basic types (Section 5.1.2.1), defined as part of projects (Section 5.1.1).
- **G2**: Definition of additional modelling elements (attributes) that may refine the basic modelling elements and allow representing additional aspects of variability. The IVML realizes this requirement by attributing modelling elements by further (orthogonal) configuration capabilities (Section 5.2.1).
- **G3**: Definition of different types of value ranges, including non-Boolean variability. The IVML realizes this requirement by enumerations (Section 5.1.2.2) and container types (Section 5.1.2.3) of the basic types (Section 5.1.2.1), derived types (Section 5.1.2.4) and compounds (Section 5.1.2.5).
- **G4**: Definition of product configurations, that allow describing individual configurations based on the definition of the configuration space. In the IVML this requirement is realized by assigning values to decision variables (Section 5.1.3), either using default values or constraints (shown in various examples in the sections above).
- **G5**: Definition of constraints to restrict the combination of elements defined in the (unconstrained) configuration space. Requirement G5 is realized by a constraint language based on OCL (Section 5.1.4), in particular by logical, relational and arithmetic expressions.

- **G6**: Definition of complex dependencies among the individual variability elements (and their attributes). Requirement G6 is realized by a constraint language based on OCL (Section 5.1.4).

Requirements for Variability Modelling in Service-Based Systems

- **S1**: Definition of the variability relevant to service platform infrastructure, technical platform services, domain-specific services, service composition and processes and service and platform deployment. This requirement is an extension of the general requirement G1. In Section 4.4, we discussed variability relevant to these variability objects. The result of this discussion was that the need to address arbitrary and unforeseeable extension to base functionality is a major challenge. This is subsumed by requirement S15. Requirement S15 is realized by inclusion and composition of projects (Section 5.2.3.2) and refinement of compounds (Section 5.2.2.1). Evolution of configuration spaces is further supported by versions and the modelling of explicit conflicts (Section 5.2.3.1).
- **S2**: Definition of (at least) the following forms of variation: optional, alternative and multiple selection. In the IVML, primary means to realize S2 are Boolean decision variables (Sections 5.1.2.1 and 5.1.3), enumerations (Section 5.1.2.2)and container (Section 5.1.2.3).

Large Scale Variability

- **S3**: Grouping of related variability elements, e.g. to define a set of alternative services from which the selected variant must be configured individually. This requirement is an extension of the general requirement G1 and helps to organize large scale variability. Requirement S3 is realized in the IVML by compound types (Section 5.1.2.5).
- **S4**: Definition of "arrays" to represent sets of cases that need to be configured while the relevant configuration possibilities have the same structure. Arrays correspond in the IVML to sequences (Section 5.1.2.3).
- **S5**: Definition of strong dependencies in the sense that general (abstract) decisions lead to the configuration of (multiple) lower-level decisions. This requirement is realized in the IVML by defining constraints (Section 5.1.4) that restrict a (set of) decision variable based on the values of other decision variables.

Quality of Service

- **S6**: Specification of qualitative and quantitative properties describing QoS for individual configurable elements. In the IVML, QoS properties can be modelled as decision variables (Section 5.1.3) or as part of compounds (Section 5.1.2.5). Qualitative properties can be modelled as (ordered) enumerations (Section 5.1.2.2) and quantitative properties as decision variables of type **`Real`** (Section 5.1.2.1).
- **S7**: Specification of arithmetic expressions to specify derived quality properties (an example could be the overall quality of a service to be computed from responsiveness and availability). In the IVML, arithmetic expressions are

supported by the constraint language (Section 5.1.4) which also supports the definition of named arithmetic expressions in the sense of function macros.

- **S8**: Definition of Constraints on QoS properties to express valid ranges and dependencies among QoS. Constraints may be used (in combination with arithmetic expressions) to map between qualitative and quantitative properties. This requirement extends the general requirements G5 and G6. The constraint language in the IVML supports the specification of valid ranges and dependencies (Section 5.1.4) on decision variables which can be used to model QoS properties (see fulfilment of requirement S6).

Meta-Variability

- **S9**: Definition of meta-variabilities and their individual properties. This requirement is an extension of the general requirements G1 and G2. Requirement S9 is realized by attributing already defined modelling elements (Section 5.2.1).
- **S10**: Definition of the impact meta-variabilities have on the implementation process. This requirement is an extension of the general requirement G4. This requirement is realized by value assignments to attributes, either as default values (Section 5.2.1) or by using value assignments of the constraint language (Section 5.1.4).
- **S11**: Definition of constraints on meta-variabilities, e.g. to express that configuring the binding time of one configuration space element restricts the binding time of a dependent configuration space element. This requirement is an extension of the general requirements G5 and G6. Requirement S11 is realized in the IVML by supporting constraints (Section 5.1.4) on attributes (Section 5.2.1).

Eco-Systems

- **S12**: Composition of multiple variability models as well as their individual configuration and the configuration across variability model boundaries. This requirement is realized in the IVML by importing (multiple) source projects into one target project (Section 5.2.3.2), thus making the type definitions, the defaults, constraints and assigned decision values part of the target project.
- **S13**: Definition of constraints among modelling elements of different variability models including a clear identification of each element with respect to its source. In the IVML, each modelling element can be accessed in terms of its qualified name (Section 5.2.3.2), in particular as parts of constraints (Section 5.1.4).
- **S14** Partial configuration of variability models to enable pre-configuration and reuse of existing variability models in new contexts. This requirement is realized in the IVML by default values (Section 5.1.3) and value assignments as part of constraints (Section 5.1.4). Further, the IVML supports freezing configured decision variables (Section 5.2.4.2) as well as partial evaluation (Section 5.2.4.3) in order to ease handling partial configurations.
- **S15**: Extension of the configuration space by variabilities that have not been taken into account previously (open variation). This can be seen as a special case of S12, however, it goes beyond it by demanding that existing variations can also be extended at a later point. Requirement S15 is realized by inclusion and composition of projects (Section 5.2.3.2) and refinement of compounds (Section

5.2.2.1). Evolution of configuration spaces is further supported by versions and explicit conflicts (Section 5.2.3.1).

- **S16**: Separation between local and global variability implementation reusing variability models (modularity). In particular, this leads to the requirement of variability interfaces. This requirement is realized by (multiple) interfaces for IVML projects (Section 5.2.3.3).

Requirements from Industry

- **I1**: Definition of non-Boolean variability. This is subsumed by requirement G3. This requirement is realized by the various types for decision variables introduced in Section 5.1.2.
- **I2**: Definition of cardinality (i.e., making – restricted – choices in groups of elements and replicate complete groups in order to provide individual configuration possibilities for the various copies. This can be seen as an extension to the requirement S3. Requirement I2 is realized in the IVML by constraint operations on collections, particularly those representing first-order logic quantors (Section 5.1.4).
- **I3**: Referencing of other (configuration) elements. Requirement I3 is realized by qualified names of configuration elements and, in particular, the reference type (Section 5.2.2.2) which realizes configuration references.
- **I4**: Integration with existing domain-specific languages. Due to its specialized nature, we consider this as a low-priority requirement, but analyse this further in Section 3.6. The IVML supports the specification and embedding of DSL code, passing configured decision values to a DSL interpreter and using the result of the DSL interpretation in the IVML (Section 5.2.5). This restricts the kinds of possible DSL-integrations. We are further analysing the need for DSL-integration, its specific requirements and further ways to address this.
- **I5**: Automatic deduction of lower-level configuration choices from higher-level configuration selections. This is subsumed by requirement S5. This requirement is realized in the IVML by defining constraints (Section 5.1.4) that restrict a (set of) decision variable based on the values of other decision variables.
- **I6**: Definition of numerical constraints. This is subsumed by requirements G6, S7, S8. In the IVML this is realized by relational and arithmetic constraints (Section 5.1.4)
- **I7**: Definition of binding-time dependencies of constraints. This is implicitly handled in the IVML according to the point of time when models are composed, i.e. when the definition of the constraints becomes available. This may also happen at later binding-times, e.g. deployment or even runtime.
- **I8**: Definition of different views (according to different criteria) of the variability model to reduce the complexity a user has to deal with. This requirement is realized by supporting the definition of different interfaces of projects (Section 5.2.3.3). Each interface can be seen as a different view on the project exporting the corresponding modelling elements.
- **I9**: Working with defaults, including complete default profiles describing defaults. This may be handled as well on the tool level. The IVML realizes this requirement by supporting the definition of default values for decision variables (Section 5.1.3) including single variables and compounds. The variables of an entire

project may have default values representing an entire default profile. In addition, a project may derive from another one by only importing it and assigning default values. Thus, the project implements a default profile.

- **I10**: Integrated configuration of modularized (and composite) product lines. This is mostly subsumed by S12 and S15. The IVML integrates modelling and configuration of projects (Section 5.1.1) which, in particular, applies to modularized projects, i.e. projects which define at least one interface (Section 5.2.3.3), and composition of projects (Section 5.2.3.2).
- **I11**: Integration with legacy product lines. In particular, the harmonized configuration of different product lines. This is mostly subsumed by S12 and S15. This requirement is realized by supporting the import and composition of projects (Section 5.2.3.2), the use of the imported modelling elements as well as the configuration of all available (unfrozen) decision variables with respect to their constraints.

5.4 Summary

In the previous sections we described the concepts of the INDENICA Variability Modelling Language (IVML). The descriptions of the concepts included the syntax of the concepts, the provided keywords that are required to define the IVML elements and expressions, as well as detailed syntax descriptions and examples. Finally, we provided a summary on how the requirements collected in Section 2 are implemented by the IVML, i.e. to which degree individual requirements are realized by the language.

The IVML provides for stepwise enhancement of expressiveness by means of core language and advanced modelling concepts. The concepts of the core modelling language are based on the results of the discussion in Section 3, while the concepts of the advanced modelling language rely on the discussion in Section 4. The INDENICA variability modelling core language supports basic and compound types, including the definition of individual or derived types, for defining decision variables as well as a corresponding set of constraints for dependency management. These concepts are sufficient for basic variability modelling as described in Section 3. The advanced modelling concepts provide capabilities like the attachment of additional attributes to modelling elements, extension mechanisms, partial configuration, etc. These concepts address the specific requirements for variability modelling in service-based systems and, in particular, in the INDENICA project as described in Section 4.

In addition, the IVML provides a lightweight concept for including domain-specific languages (DSLs) as part of the variability model. This is to support situations, in which the variability may be expressed more intuitively or more naturally using DSLs.

The summary on how the requirements collected in Section 2 are implemented by the IVML shows that this variability modelling language completely supports the required modelling capabilities for the INDENICA project. However, further enhancements may occur to increase the ease of using the IVML. If they will occur, they will be shown in future deliverables (D2.2.2).

6 Application of IVML in the INDENICA Case Studies

This section analyses the application of IVML concepts in the industrial case studies described in Deliverable D5.1. We will consider (preliminary) variability models created by partners during early evaluations of IVML as well as models developed during implementation of the case studies. As part of the analysis, we will discuss semantically meaningful fragments of the variability models of the three INDENICA case studies. This section is organized in a similar order to the introduction of IVML concepts in section 5, i.e., we will first show the usage of the basic concepts in Section 6.1 and then the application of advanced concepts in Section 6.2. Finally, in Section 6.3 we will summarize the overall application of IVML concepts in the three INDENICA case studies.

6.1 Application of the Core Language

In this section, we will discuss fragments from the variability models of the INDENICA case studies, which illustrate the application of the core concepts of IVML. We will start with the Remote Maintenance case as the structure of the model allows to easily present a fragment representing an entire IVML project. Then we will discuss fragments from the Warehouse and the Yard management case.

```
project Remote_Maintance {
    enum ServiceState {full, point2point, off};

    ServiceState videoCalls = ServiceState.point2point;
    ServiceState audioCalls = ServiceState.full;
    Boolean changeUsersInCall = true;

    videoCalls = off and audioCalls = off
      implies changeUsersInCall = false;
}
```

Figure 10: Fragment from the variability model of the Remote Maintenance case study.

Figure 10 shows a fragment of the variability model for configuring a virtual machine in the Remote Maintenance Case. Figure 10 illustrates the **declaration of a project** (cf. Section 5.1.1), the **definition of an enumeration** (cf. Section 5.1.2.2) as well as the **declaration of decision variables** (cf. Section 5.1.3) using the previously defined enumeration and the **basic type** Boolean (cf. Section 5.1.2.1). While `changeUsersInCall` is a Boolean decision variable (optional selection), `videoCalls` and `audioCalls` represent alternative selections among three states. The fragment in Figure 10 also illustrates a **constraint** on the defined decision variables (cf. Section 5.1.4). This constraint states that if the capabilities for `videoCalls` and `audioCalls` are disabled, changing users in a call (`changeUsersInCall`) is not supported.

Figure 10 explicitly depicts the definition of the containing project. However, in the fragments below we will concentrate on the specific use of selected IVML concepts and will not provide a project definition, unless it is explicitly required.

```
typedef forkNumber Integer with (forkNumber >= 0
    and forkNumber <= 8);

compound rackOperatorType {
   Integer maxSpeed;
   forkNumber forkCount;
}

compound laneType {
   Integer maxX, maxY;
}

compound highRack {
   sequenceOf(laneType) rackLanes;
}
```

Figure 11: Simplified fragment from the WMS variability model.

Figure 11 shows a fragment from the variability model of the Warehouse Management System (WMS) case study[29]. This fragment defines the constrained Integer type `forkNumber` using **type derivation** (cf. Section 5.1.2.4). The type derivation, in turn, puts a constraint on all the decision variables of type `forkNumber`, e.g., on the decision variable `forkCount` in the **compound** (cf. Section 5.1.2.5) `rackOperatorType`. Further, the compound `highRack` defines a decision variable `rackLanes` as a **collection** (cf. Section 5.1.2.3) of instances of the compound `laneType`.

The WMS variability model particularly aims at specifying the topology of warehouses supported by the underlying WMS and to configure the specific topology of a certain warehouse at a customer site. This is achieved by combining decision variables of different compound types, each capturing information of a specific aspect of a warehouse. In Figure 11 this is illustrated in terms of the individual compound declarations and their nesting using a container decision variable.

Please note that Figure 11 is a fragment for illustrating the use of the core concepts of IVML in the context of the WMS case study. In particular, constraints and relationships among the individual types defined in that model are not discussed in this section as they rely on advanced modeling concepts although these relationships are important for modeling and finally configuring the topology of a certain warehouse.

[29] Please note that the original model uses German terminology. In order to support readability in this deliverable, we translated the terms to English.

```
enum Location {cell, gps};
enum SchedulingType {simple, locationBased};

compound yjs {
    SchedulingType scheduling;
    Location location;
}

sequenceOf(yjs) ModuleList;
Boolean gps;

gps implies ModuleList->forAll(t |
    t.location = Location.gps);
```

Figure 12: Simplified fragment form the YMS case study.

Figure 12 depicts a fragment from the variability model of the Yard Management System (YMS) case study. At first glance, it uses similar core IVML concepts as the fragments shown above such as the definition of **enumerations** (cf. Section 5.1.2.2), of **compounds** (cf. Section 5.1.2.2) and **decision variables** (cf. Section 5.1.4) using **basic types** (cf. Section 5.1.2.1) as well as **container types** (cf. Section 5.1.2.3).

In contrast to the fragments shown above, this model specifies the variability within subsystems of the YMS. In Figure 12, a part of the variability of the Yard Jockey Support (`yjs`) subsystem is shown. Yard jockeys are informed about new tasks according to a `scheduling` mode, which may either lead to `simple` task assignments or to `locationBased` assignments. The relationship among the alternatives expressed by the decision variables of `yjs` and their individual types is further specified by a constraint shown at the bottom of Figure 12. This **constraint** states that if `gps` (given as a global decision variable) is enabled, the `location` mode of all decision variables of compound type `yjs` in `ModuleList` (**quantor expression**) must be `gps` (defined by the enumeration `Location`).

```
project WMS_Configuration {
    autoLaneType wms;
    wms = {maxX = 5,
           maxY = 7,
           rackOperatorType = {maxSpeed = 42,
                               forkNumber = 5}};
}
```

Figure 13: Configuration of decision variables in the WMS case study.

All three industrial use cases apply the **configuration of decision variables** (cf. Section 5.1.5). An example configuration of the WMS model is shown in the fragment in Figure 13. The fragment illustrates the configuration of the decision variable `wms`, which is declared as a variable of type `autoLaneType`.

During the initial evaluations of IVML by the INDENICA partners it was highlighted that such configurations, which exclusively rely on domain terminology appear natural to domain experts.

6.2 Usage of Advanced Concepts

In this section, we will discuss the application of the advanced modelling concepts of IVML in the INDENICA case studies. Following the structure of Section 5.2, we will start with an example for attributes, continue with versioned project composition, and end this section with compound extension.

```
project Remote_Maintenance {
    version 0.5;

    enum BindingTimes {compile = 0, deployment = 1,
        runtime = 2};

    attribute BindingTimes bindingTime
        = BindingTimes.compile to Remote_Maintenance;

    // contents from Figure 10 follows here

    enum ServiceState {full, point2point, off};

    ServiceState videoCalls = ServiceState.point2point;

    // …
}
```

Figure 14: Fragment of the compiletime variability model from the Remote Maintenance case study.

In INDENICA, variability models will be instantiated at different times throughout the software lifecycle (binding time), in particular at compile time, deployment time, and runtime. Currently, the YMS as well as the Remote Maintenance use case specify variability with different binding times. Below, we will illustrate the usage of meta-variabilities, more specifically binding times, using the IVML concept of **attributes** (cf. Section 5.2.1).

Figure 14 depicts a fragment from the Remote Maintenance case study, more precisely, Figure 14 illustrates the compile time part of the variability model. The entire variability model of the Remote Maintenance case consists of multiple projects, which are finally composed to a single variability model.

The project `Remote_Maintenance` shown in Figure 14 defines an **ordered enumeration** of binding times, i.e., an implicit partial order of the enumeration literals so that compile time can be considered as a predecessor of deployment time, which, in turn, acts as a predecessor of runtime. Further, it defines an attribute called binding time with **default value** `compile` (from the previously defined enumeration) and attaches this attribute to the entire project, i.e., also to all contained decision variables. This attribute will be considered during instantiation, i.e., decision variables attributed for compile time must be

configured before the related variability instantiation mechanism will modify any generic artefacts.

In addition, the project Remote_Maintenance shown in Figure 14 declares its own **version** (here 0.5, cf. Section 5.2.3.1) which will be relevant when we now discuss the application of project compositions in the INDENICA case studies.

```
project Remote_Maintenance_Runtime {

    import Remote_Maintenance with
        (Remote_Maintenance.version >= 0.5);

    attribute BindingTimes bindingTime
       = BindingTimes.runtime to Remote_Maintenance_Runtime;

    compound ServerState {
        Real freeMemory;
        Real cpuUsage;
        Integer availableVMs;
    }

    Integer currentUsers;
    Integer currentVMs;

    ServerState currentServerState;
    Boolean serversAtHalfFullSpeed;
    Integer additionalVMs;

    serversAtHalfFullSpeed =
        currentServerState.freeMemory < 0.2
        or currentServerState.cpuUsage > 0.8;

    currentUsers / currentVMs > 10000
      implies additionalVMs > 0;

}
```

Figure 15: Fragment of the composed runtime variability model from the Remote Maintenance case study.

Figure 15 depicts a fragment of the runtime variability model part of the Remote Maintenance case study. Please note that all industrial partners modularized their models into multiple projects for different reasons such as building small and understandable units or grouping together variables of the same binding time.

The project Remote_Maintenance_Runtime shown in Figure 15 imports another project through **project composition** (cf. Section 5.2.3.2), here the compile time part. Further, it also defines an attribute denoting the binding time for the project and its contents, here with runtime as default binding time. Please note that the enumeration BindingTimes is not redefined in this project as it is available through the import of Remote_Maintenance. Further, please note that the import of Remote_Maintenance is constrained by a minimum **version number** (cf. Section 5.2.3.1). This is due to the fact that previous versions of the variability

model for the Remote Maintenance use case did not contain binding times, i.e., the enumeration `BindingTimes` was not defined, and so the use of these outdated models must be prevented.

The project `Remote_Maintenance_Runtime` defines several decision variables to be resolved at runtime, such as the server state, overall numbers of virtual machines in use or even the actual number of users working on the platform. In addition, the first constraint influences the speed of the servers at runtime (`serversAtHalfFullSpeed`) while the second constraint requires (some) addition virtual machines (`additionalVMs`) due to an **arithmetic calculation** of a derived metric.

```
typedef forkNumber Integer with (forkNumber >= 0
    and forkNumber <= 8);

compound rackOperatorType {
    Integer maxSpeed;
    forkNumber forkCount;
}

compound laneType {
    Integer maxX, maxY;
}

compound autoLaneType refines laneType {
    rackOperatorType operatorDevice;
}
compound highRack {
   sequenceOf(laneType) rackLanes;
}
```

Figure 16: Compound refinements in the WMS variability model.

Both, the WMS and the YMS case study rely on the **extension of compounds** (cf. Section 5.2.2.1). Figure 16 illustrates compound refinements in the WMS model. Here, the compound `autoLaneType` refines the compound `laneType`, i.e., lanes with automated rack operators (`autoLaneType`) are also lanes (`laneType`) but provide additional variability. In IVML, extended compounds are similar to subclasses in object orientation, i.e., `autoLaneType` is also of type `laneType` and, thus, instances of `autoLaneType` may be used to define the `rackLanes` of a `highRack`. (The difference is of course that as opposed to object orientation no methods can be defined as IVML does not directly support the modeling of activities.) If not further restricted by constraints, a `highRack` may consist of both kinds of lanes (heterogeneous collection). In addition, future versions of the WMS may even provide new types of lanes with individual variability (open-world scenario). This can easily be integrated into existing variability models through extension and project composition.

Finally, the configuration of a variability model is used for instantiating the underlying product line artefacts. Instantiation mechanisms will take the

configuration as an input and modify generic artefacts in the solution space accordingly. However, configurations may change along project compositions, i.e., some imported projects may define a preconfigured profile of a platform, which is (partially) overridden later. For the final instantiation step, only stable configurations can be considered, i.e., those decision variables which are **frozen** (cf. Section 5.2.4.2). We will demonstrate this in terms of the WMS fragment already shown in Figure 13.

```
import WMS_Configuration;
freeze {
    wms;
}
```

Figure 17: Freezing configuration values in the WMS case study.

Figure 17 imports the (partial) configuration from the project shown in Figure 13. Thereby, the imported project is implicitly evaluated and `wms` receives a value, which is, however, only a default value. In order to use it as a basis for the final instantiation, the decision variable `wms` is frozen as shown in Figure 17. However, having a configuration project and a freezing project is not always a practical approach, in particular for tool support. Thus, a semantically equivalent model can be expressed using the **partial evaluation** mechanism (cf. Section 5.2.4.3).

```
import WMS;
eval {
    wms = {maxX = 5,
           maxY = 7,
           rackOperatorType = {maxSpeed = 42,
                               forkNumber = 5}};
}
freeze {
    wms;
}
```

Figure 18: Combination of partial evaluation and freezing a configuration in the WMS case study.

Figure 18 depicts the combination of partial evaluation and freezing variables in one project. The fragment imports the WMS variability model (an extension of Figure 12), evaluates and then freezes the configuration. This combination of partial evaluation with freezing is required as IVML does not provide support for defining the order of evaluation, i.e., without partial evaluation it is not guaranteed the configuration is defined before freezing the variables. As partial evaluations are executed inside-out according to their nesting, first the partial evaluation block, i.e., the configuration is processed and then the configuration is frozen as specified.

Currently, the configurations of the variability models in the use case studies use either the first or the second approach to specify their final configuration for instantiation.

6.3 Overview on IVML Concepts used in INDENICA case studies

In this section, we summarize the IVML concepts used in the INDENICA case studies so far in terms of a mapping of concepts to the case studies. Please note that we discussed only fragments of these models in the sections above. This mapping of language concepts to case studies is summarized in Table 3.

IVML-Level	IVML Concept	YMS	WMS	Remote Maintenance
Core Language	Project	x	x	x
	Basic types: Boolean, Integer, Real, String	Integer, String, Boolean	Integer	Integer, Real, Boolean, String
	Enumerations	x	-	x
	Container	x	x	-
	Type derivation and restriction	-	x	-
	Compounds	x	x	x
	Decision variables	x	x	x
	Constraints	x	x	x
	Configurations	x	x	x
Advanced	Attributes	x	-	x
	Extended compounds	x	x	-
	Referenced elements	-	-	-
	Project versioning	-	-	x
	Project composition	x	x	x
	Project interfaces	-	-	-
	Partial configuration	-	-	-
	Freezing configurations	x	x	x
	Partial evaluation	x	x	-
	DSL inclusion	-	-	-

Table 3: Overview of currently used IVML concepts in INDENICA case studies.

Although the variability models as well as the case study implementations are still under development, all the core IVML concepts as well as most of the advanced language concepts are already used in the INDENICA case studies.

However, there are some of the advanced language concepts, which are currently not used in any case study. This is due to the fact that the case studies and their variability models are currently under development and the models will evolve until the end of the INDENICA project. We expect that modularization and model composition will be used extensively when the complexity and size of the models increase. For example, the WMS model is already modularized into small parts with related constraints but information hiding through **interfaces** is yet not established. Further, the partners aim at creating a global variability model for all three service

platforms in order to enable cross-platform variability including constraints. We expect that this will lead to the use of interfaces as the internals of the individual variability models shall not be visible in the global model in order to simplify modelling as well as configuration. As part of these activities, also **partial configuration** will be used as a natural extension of already used configuration concepts. Moreover, we also expect the use of **configuration references**, in particular in the WMS case study, as this concept specifically supports the creation of networks and the specification of complex topologies.

While we are confident about the advanced concepts mentioned above, we also see that the need for the **inclusion of domain-specific languages** decreased since the requirements elicitation phase for IVML. One particular feedback from our industrial partners in applying IVML is that textual variability configurations appear natural to domain experts as also mentioned in Section 6.1. Thus, we decreased the priority for DSL inclusion during the design phase of IVML and provide just a basic mechanism to realize requirement I4.

It should also be emphasized that even if some of the concepts will not be exploited in the final use case studies, this does not necessarily mean that these concepts were superfluous or inappropriate. For example configuration references were introduced based on a different use case provided by Siemens, which was intensively discussed, but later the decision was made that these case studies should not be further pursued in favour of the finally chosen ones. It was our aim with the IVML to provide a language that ideally covers the challenges of service platform configuration as broad as possible. Thus, the absence of some concepts in some case studies does not lead to the inappropriateness of the corresponding language in the same way as the fact that some language elements are not used in a specific program is no indication of the inappropriateness of this program language.

What we see as the main evaluation criterion is that language is able to cover all relevant cases. Here, the experiences in the industrial case studies were very positive. We also discussed the language with other industrial partners outside the project and got very positive feedback as well. Thus, we currently assume that this language is very well suited for the configuration of service platforms.

7 Conclusion

In this deliverable, we aimed at determining the core expressiveness of the INDENICA variability modelling language and further extensions that arise from the specific requirements of the INDENICA project. As a consequence, the structure of this deliverable followed the distinction of core modelling aspects and advanced variability modelling extensions.

In a first step, we identified and defined the requirements for variability modelling in INDENICA. This information was derived from multiple sources, including general variability modelling requirements, demands for variability modelling in service-based systems and feedback from the industrial partners in the project. The set of requirements served as a basis for the discussions and decisions towards the concepts of the INDENICA variability modelling language.

In Section 3, we discussed core variability modelling concepts. The focus was to categorize different variability modelling concepts with respect to their expressiveness. This discussion covered the supported types of modelling elements and the provided dependency management capabilities. On this basis, we identified the required core expressiveness of the INDENICA approach to satisfy general variability modelling requirements as well as to provide a basis for the advanced modelling concepts in an effective and easy to use manner.

Section 4 focused on advanced variability modelling concepts to provide an overview of additional extensions to core variability modelling that are required by the INDENICA project. Again, we discussed the required modelling elements and dependency management capabilities for modelling, e.g. Quality of Service (QoS) and meta-variabilities. We emphasize concepts like modularity and extensibility as these are mayor issues, in particular, in service and service platform ecosystems. Some identified concepts were already covered by the concepts we demand for the INDENICA core modelling language in Section 3 (e.g. non-Boolean variability to express quantitative properties for QoS).

Section 5 then introduced the concepts of the INDENICA variability modelling language. These concepts were selected based on the results of our previous discussions in Section 3 and Section 4. The main contribution is to provide a variability modelling language that satisfies all identified requirements in Section 2 in an effective and easy to use manner. The core modelling concepts of the INDENICA variability modelling language enable the modelling of basic variabilities. The advanced modelling concepts extend the core concepts to satisfy the specific demands that arise in the INDENICA project. Again, ease of use for the most standard issues was important, so we extended the core concepts in a way so that their use is not complicated for users who do not need the more advanced features.

Section 6 analyzed the application of IVML variability modeling concepts in the industrial case studies in INDENICA. Therefore, we discussed fragments of the variability models created for the base platforms by the industrial partners in terms of their support for core language concepts and advanced language concepts. We

summarized the application of concepts and showed that all basic concepts as well as almost all advanced concepts of IVML are used in the case studies. For concepts, which are currently not used, we discussed their expected use until the end of the project. In summary, we are confident that all concepts except for DSL inclusion will be used in INDENICA. Moreover, the experience in the project so far is that all we found basically all relevant concepts based on our detailed requirements analysis. We thus expect that there will be only rather minor extensions to the language till the end of the project, if any.

Further work on the IVML will focus on the deep integration with the concepts discussed in deliverable D2.2.1 and the systematic evaluation based on the project case studies.

References

[1] M. Acher, P. Collet, P. Lahire, and R. B. France. A Domain-Specific Language for Managing Feature Models. In *Proceedings of the 26th ACM Symposium on Applied Computing (SAC '11)*, 2011.

[2] O. Aoki. Debian Reference, 2007. Online available at: http://qref.sourceforge.net/Debian/reference/reference.en.pdf.

[3] D. Ayed and Y. Berbers. Dynamic Adaptation of CORBA Component-Based Applications. In *Proceedings of the 22nd Annual ACM Symposium on Applied Computing (SAC '07)*, pages 580–585, 2007.

[4] K. Bak, K. Czarnecki, and A. Wasowski. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In B. Malloy, S. Staab, and M. van den Brand, editors, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE '10)*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2010.

[5] J. Bartholdt, M. Medak, and R. Oberhauser. Integrating Quality Modeling with Feature Modeling in Software Product Lines. In K. Boness, J. M. Fernandes, J. G. Hall, R. J. Machado, and R. Oberhauser, editors, *Proceedings of the 4th International Conference on Software Engineering Advances (ICSEA '09)*, pages 365–370. IEEE Computer Society, 2009.

[6] D. Batory, D. Benavides, and A. Ruiz-Cortes. Automated Analysis of Feature Models: Challenges Ahead. *Communication of the ACM (CACM)*, 49(12):45–47, 2006.

[7] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated Reasoning on Feature Models. In *Proccedings of the 17th International Conference on Advanced Information Systems Engineering (CAISE '05)*, pages 491–503, 2005.

[8] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE '05)*, pages 677–682, 2005.

[9] D. Benavides, S. Segura, and A. Ruiz-Cortes. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35:615–636, 2010.

[10] G. Botterweck, A. Polzer, and S. Kowalewski. Interactive Configuration of Embedded Systems Product Lines. In *Proceedings of the 1th International Workshop on Model-driven Approaches in Product Line Engineering (MAPLE'09), collocated with the 13th International Software Product Line Conference (SPLC'09)*, 2009.

[11] G. Botterweck, A. Polzer, and S. Kowalewski. Using Higher-order Transformations to Derive Variability Mechanism for Embedded Systems. In

Proceedings of the 12th International Workshop and Symposia on Model Driven Engineering Languages and Systems (MoDELS'09, 2009.

[12] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-Based Feature Modelling Language. In *Proceedings of the 4th International Workshop on Variabilits Modelling of Software-Intensive Systems (VaMoS '10)*, pages 159–162, 2010.

[13] H. Brummermann, M. Keunecke, and K. Schmid. Formalizing Distributed Evolution of Variability in Information System Ecosystems. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems (VaMoS '12)*, 2012.

[14] W. Cazzola and D. Poletti. DSL Evolution through Composition. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE '10)*, RAM-SE '10, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[15] K. Chan and I. Poernomo. QoS-Aware Model Driven Architecture through the UML and CIM. In *Proceedings of the 10th IEEE International Enterprise Distributed Object Computing Conference (EDOC '06)*, pages 345–354, December 2006.

[16] M. Clauß. Generic Modeling Using UML Extensions for Variability. In *Proceedings of the Workshop on Domain Specific Visual Languages*, pages 11–18, 2001.

[17] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool Features and Tough Decisions: Two Decades of Variability Modeling. In *Proceedings of the 6th International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '12)*, 2012. To appear.

[18] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Using Feature Models. In Robert Nord, editor, *Software Product Lines*, volume 3154 of *Lecture Notes in Computer Science*, pages 162–164. Springer, 2004.

[19] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-Based Feature Models and their Specialization. *Software Process: Improvement and Practice, Special Issue on Software Variability: Process and Management*, 10(1):7 – 29, 2005.

[20] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged Configuration Through Specialization and Multi-Level Configuration of Feature Models. *Software Process Improvement and Practice*, 10(2):143–169, 2005. Special Issue on Software Product Lines.

[21] K. Czarnecki and P. Kim. Cardinality-Based Feature Modeling and Constraints: a Progress Report. In *Proceedings of the International Workshop on Software Factories at the 10th International Conference on Object-Oriented Progamming, Systems, Languages and Applications (OOPSLA '05)*, 2005.

[22] T. M. Dao, H. Lee, and K. C. Kang. Problem Frames-Based Approach to Achieving Quality Attributes in Software Product Line Engineering. In de Almeida et al. [23], pages 175–180.

[23] E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, editors. *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*. IEEE, 2011.

[24] D. Dhungana, P. Grünbacher, and R. Rabiser. The DOPLER Meta-Tool for Decision-Oriented Variability Modeling: a Multiple Case Study. *Automated Software Engineering*, 18:77–114, 2011.

[25] S. Dustdar and F. Li. *Service Engineering - European Research Results*. Springer, 2011.

[26] S. El-Sharkawy, C. Kröher, and K. Schmid. Supporting Heterogenous Compositional Multi Software Product Lines. In *Proceedings of the 15th International Software Product Line Conference (SPLC '11)*, volume 2, 2011.

[27] C. Elsner, P. Ulbrich, D. Lohmann, and W. Schr/öder-Preikschat. Consistent Product Line Configuration Across File Type and Product Line Boundaries. In *Proceedings of the 14th International Software Product Line Conference (SPLC '10)*, pages 181–195, 2010.

[28] European Software Institute Spain, IKV++ Technologies AG Germany. *MASTER: Model-driven Architecture inSTrumentation, Enhancement and Refinement*, ist-2001-34600, master-2002-d1.1-v1-public edition, 2002.

[29] M. Fowler and R. Parsons. *Domain-Specific Languages*. Addison Wesley Signature Series. Addison-Wesley, 2010.

[30] S. Frolund and J. Koistinen. QML: A Language for Quality of Service Specification. Technical Report HPL-98-10, Hewlett Packard Company, 1998.

[31] C. Ghezzi and A. Molzam Sharifloo. Verifying Non-Functional Properties of Software Product Lines: Towards an Efficient Approach Using Parametric Model Checking. In de Almeida et al. [23], pages 170–174.

[32] D. Ghosh. *DSLs in Action*. Manning Publications, 1 edition, December 2010.

[33] D. Ghosh. DSL for the Uninitiated. *Queue*, 9:10:10–10:21, June 2011.

[34] H. Haas and A. Brown. World Wide Web Consortium (W3C) Web Service Glossary: Quality of Service, 2011. Online available at: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211.

[35] S. Hallsteinsen, E. Stav, A. Solberg, and J. Floch. Using Product Line Techniques to Build Adaptive Systems. In *Proceedings of the 10th International Software Product Line Conference (SPLC'06)*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.

[36] O. Haugen, B. Møller-Pedersen, J. Oldevik, G. K. Olsen, and A. Svendsen. Adding Standardized Variability to Domain Specific Languages. In *Proceedings of the 12th International Software Product Line Conference (SPLC '08)*, pages 139–148, Washington, DC, USA, 2008. IEEE Computer Society.

[37] G. Holl, M. Vierhauser, W. Heider, P. Grünbacher, and R. Rabiser. Product Line Bundles for Tool Support in Multi Product Lines. In *Proceedings of the 5th*

International Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11), pages 21–28, 2011.

[38] D. Jackson, I. Schechter, and H. Shlyahter. Alcoa: The Alloy Constraint Analyzer. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, ICSE '00, pages 730–733, New York, NY, USA, 2000. ACM.

[39] K. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A Feature-Oriented Reuse Method with Domain-Specific Reference Architecture. *Annals of Software Engineering*, 5:143–168, 1998.

[40] 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 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University, 1990.

[41] A. Keller and H. Ludwig. The WSLA Framework: Specifying and Monitoring Service Level Agreements for Web Services. *Journal of Network and Systems Management*, 11:57–81, March 2003.

[42] H. Krahn, B. Rumpe, and S. Völkel. Efficient Editor Generation for Compositional DSLs in Eclipse. In *Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM' 07)*, Montreal, Quebec, Canada, 2007.

[43] D. D. Lamanna, J. Skene, and W. Emmerich. SLAng: a Language for Defining Service Level Agreements. In *Proceedings of the 9th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '03)*, FTDCS '03, pages 100–, Washington, DC, USA, 2003. IEEE Computer Society.

[44] Y. Li, X. Zhang, Y. Yin, and J. Wu. QoS-Driven Dynamic Reconfiguration of the SOA-Based Software. In *Proceedings of the 2010 International Conference on Service Sciences (ICSS '10)*, pages 99–104, 2010.

[45] A. Ludwig and B. Franczyk. COSMA — An Approach for Managing SLAs in Composite Services. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC '08)*, ICSOC '08, pages 626–632. Springer, 2008.

[46] J. X. Mansell and D. Sellier. Decision Model and Flexible Component Definition Based on XML Technology. In *Proceedings of the 5th International Workshop on Software Product-Family Engineering (PFE 2003)*, pages 466–472, 2003.

[47] R. Mietzner, A. Metzger, F. Leymann, and K. Pohl. Variability Modeling to Support Customization and Deployment of Multi-Tenant-Aware Software as a Service Applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems (PESOS '09)*, pages 18–25, 2009.

[48] B. Mohabbati, D. Gasevic, M. Hatala, M. Asadi, E. Bagheri, and M. Boskovic. A Quality Aggregation Model for Service-Oriented Software Product Lines Based on Variability and Composition Patterns. In G. Kappel, Z. Maamar, and H. R. Motahari-Nezhad, editors, *Proceedings of the 9th International Conference on Service Oriented Computing (ICSOC '11)*, volume 7084 of *Lecture Notes in Computer Science*, pages 436–451. Springer, 2011.

[49] B. Morin, G. Perrouin, P. Lahire, O. Barais, G. Vanwormhoudt, and J.-M. Jezequel. Weaving Variability into Domain Metamodels. In *Proceedings of the 12th International Conference on Model Driven Engineering Language and Systems (MoDELS'09)*, 2009.

[50] C. Müller, O. Martn-Dáz, A. Ruiz-Cortés, M. Resinas, and P. Fernández. Improving Temporal-Awareness of WS-Agreement. In *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC '07)*, ICSOC '07, pages 193–206. Springer, 2007.

[51] Object Management Group, Inc. (OMG). Meta Object Facility (MOF) Core Specification, v2.0. Technical Report formal/06-01-01, January 2006. Online available at: http://www.omg.org/spec/MOF/2.0/.

[52] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: http://www.omg.org/docs/formal/06-05-01.pdf.

[53] Object Management Group, Inc. (OMG). Unified Modeling Language: Superstructure version 2.1.2. Specification v2.11 2007-11-02, Object Management Group, November 2007. Available online at: http://www.omg.org/docs/formal/2007-11-02.pdf.

[54] Object Management Group, Inc. (OMG). A UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Beta 2. Technical Report ptc/2008-06-09, Object Management Group, June 2008. Online available at: http://www.omg.org/docs/ptc/08-06-09.pdf.

[55] Object Management Group, Inc. (OMG). Common Variability Language (CVL), 2010. OMG initial submission. Available on request.

[56] L. Passos, M. Novakovic, Y. Xiong, T. Berger, K. Czarnecki, and A. Wasowski. A Study of Non-Boolean Constraints in Variability Models of an Embedded Operating System. In *Proceedings of the 3rd International Workshop on Feature-Oriented Software Development (FOSD '11)*, 2011.

[57] T. Phan, J. Han, J.-G. Schneider, and K. Wilson. Quality-Driven Business Policy Specification and Refinement for Service-Oriented Systems. In *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC '08)*, ICSOC '08, pages 5–21. Springer, 2008.

[58] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, August 2005.

[59] Mark-Oliver Reiser. Core Concepts of the Compositional Variability Management Framework (CVM). Technical Report 2009/16, Technische Universität Berlin, 2009. Available online at http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/.

[60] C. Riedl, T. Böhmann, M. Rosemann, and H. Krcmar. Quality Aspects in Service Ecosystems: Areas for Exploitation and Exploration. In *Proceedings of the 10th International Conference on Electronic Commerce (ICEC '08)*, number 19, 2008.

[61] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '10)*, pages 123–130, 2010.

[62] K. Schmid. Variability Modeling for Distributed Development - A Comparison with Established Practice. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2010.

[63] K. Schmid and I. John. A Customizable Approach To Full-Life Cycle Variability Management. *Science of Computer Programming*, 53(3):259–284, 2004.

[64] K. Schmid, R. Rabiser, and P. Grünbacher. A Comparison of Decision Modeling Approaches in Product Lines. In *Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems (VaMoS '11)*, pages 119–126, 2011.

[65] P.-Y. Schobbens, P. Heymans, and J.-C. Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Proceedings of the 14th IEEE Requirements Engineering Conference (RE '06)*, pages 139–148, 2006.

[66] N. Siegmund, M. Rosenmüller, C. Kästner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov. Scalable Prediction of Non-functional Properties in Software Product Lines. In de Almeida et al. [23], pages 160–169.

[67] Software Productivity Consortium Services Corporation, Technical Report SPC-92019-CMC. *Reuse-Driven Software Processes Guidebook, Version 02.00.03*, November 1993.

[68] N. Steinmetz and I. Toma. D16.1v1.0 WSML Language Reference. Technical Report D16.1v1.0, ESSI WSMO working group, August 2008. http://www.wsmo.org/TR/d16/d16.1/v1.0/.

[69] D. Streitferdt, M. Riebisch, and I. Philippow. Details of Formalized Relations in Feature Models Using OCL. In *Proceedings of the 10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS '03)*, 45-54, 2003.

[70] The Eclipse Foundation. Eclipse 3.1 Documentation: Platform Plug-in Developer Guide, 2005. Online available at: http://www.eclipse.org/documentation.

[71] V. Tosic, B. Pagurek, B. Esfandiari, K. Patel, and W. Ma. Web Service Offerings Language (WSOL) and Web Service Composition Management (WSCM). In *Proceedings of the Object-Oriented Web Services Workshop (OOWS '02)*, 2002.

[72] Rob van Ommering. *Building Product Populations with Software Components*. PhD thesis, University of Groningen, 2004.

[73] M. Voelter and E. Visser. Product Line Engineering using Domain-Specific Languages. In *Proceedings of the 15th International Software Product Line Engineering Conference (SPLC '11)*, pages 70–79, 2011.

[74] J. White, B. Doughtery, and D. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software*, 82(8):1268–1284, 2009.

[75] Øystein Haugen, Birger Møller-Pedersen, Jon Oldevik, and Arnor Solberg. An MDA®-Based Framework for Model-Driven Product Derivation, 2004.

[76] Z. Ziadi, L. Hélouët, and J.-M. Jézéquel. Towards a UML Profile for Software Product Lines. In *Proceedings of the 5th International Workshop on Software Product-Family Engineering (PFE '03)*, number 3014, pages 129—139, 2004.

[77] S. Zschaler, P. Sanchez, J. Santos, M. Alferez, A. Rashid, L-Fuentes, A-Moreira, J. Araujo, and U. Kulesza. VML* – A Family of Languages for Variability Management in Software Product Lines. In *Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09)*, pages 82–102, 2009.