Engineering Virtual Domain-Specific Service Platforms

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

Variability Implementation Techniques for Platforms and Services (Final)

Creating domain-specific service platforms requires the capability of (automatically) customizing and configuring service platforms according to the specific needs of a domain. In this deliverable we address the variability implementation techniques used in INDENICA to realize this demand.

We discuss the generic tool support provided by the developed variability engineering tool and, in particular, how the generic support is turned into specific variability implementation techniques for the three INDENICA case studies. We identify lessons learned from our practical work and turn the gained experience into a set of general concepts for variability implementation techniques. The development of the Variability Implementation Language (VIL) that provides the concepts eases the application of the INDENICA variability engineering concepts and supports their general uptake into industrial practice.

Document ID:	INDENICA – D2.2.2
Deliverable Number:	D2.2.2.
Work Package:	WP2
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2013-03-31
Author(s):	SUH, SAP, SIE, TEL

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

Version History

0.1	15. January 2013	initial version
0.2	10. February 2013	Basic structure and contents
0.3	4. March 2013	Revision of the current state section
0.6	20. March 2013	Initial description of VIL
0.8	10. April 2013	Revision of the VIL
0.9	26. April 2013	Application of VIL to INDENICA case studies
1.0	30. April 2013	Final version

Document Properties

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

Table of Contents

Table of Figures

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 the realization of the customization in terms of variability implementation techniques, in particular those which are currently applied in the INDENICA case studies. This work is of course related to the modelling of variability, which is addressed in Deliverable D2.1 [10] in terms of the INDENICA Variability Modelling Language (IVML) and Deliverable D2.4.1 [15] on the realization of the variability engineering tool.

In this deliverable, the main focus is on the specific variability implementation techniques which are actually applied in the INDENICA case studies and their support by the INDENICA variability engineering tool. Based on our practical experience in realizing the variability implementation technique, we will summarize lessons learned and introduce a set of more general concepts, which we identified. Creating a language based on these concepts eases the future application and the general uptake of the INDENICA variability implementation techniques as well as their tool support.

In Section 2, we summarize the requirements for variability implementation in INDENICA. We provide this section as an introduction to variability implementation, to the specific needs in INDENICA, and, in particular, as a link to D2.2.1 where the requirements have already been collected. In contrast to D2.2.1, we detail here the individual requirements from an implementation perspective which will serve as a basis for our discussion of the tool support, the actual implementation in the case studies and as well for future conceptual improvements.

In Section 3, we will discuss the current state of the variability instantiation in INDENICA. First, we will focus on the technical framework provided by the INDENICA variability engineering tool in terms of the architecture of the related components, their application as well as the specific support for hierarchical product lines and software ecosystems. Then, we will discuss the specific strategies realized for the three INDENICA case studies, their realization in terms of algorithmic descriptions as well as illustrating examples. In addition, we will provide an outlook on additional case studies for variability implementation we performed in INDENICA as well as in related projects. Finally, we will conclude this section with a summary of the lessons learned as well as improvements for the current state of variability instantiation.

Overall, the practical work on variability implementation techniques in INDENICA fulfils the general requirements in Section 2. However, the current state can be summarized as a Java-based implementation, which needs specific implementation work for each case study. Thus, in Section 4, we will take up the experiences we made in Section 3 and propose a more generalized framework of concepts for variability implementation. This framework will ease the realization of specific variability implementations and enable a wider uptake in future.

2 Requirements for Variability Instantiation

In this section, we provide an overview of the requirements for variability implementation. Basically, this section follows the structure of the original discussion on requirements in D2.2.1 (general capabilities, customizing service platforms, domain-specific platforms, quality requirements, binding times) but highlights the individual requirements for this deliverable in the context of the INDENICA case studies.

RQ1. **Support for handling variability implementation elements:** As identified in [11, 6] the INDENICA variability instantiation approach must support

- *Optional variability:* An implementation element may or may not be part of a platform under certain circumstances. One example is the (enabled or disabled) support for mobile clients in the Yard Management (YMS) Platform demonstrated in the review meeting.
- *Alternative variability:* The decision for a variability allows choosing one out of several alternatives, such as the communication protocols in the YMS platform [13, p. 33] or the connectivity mechanism in the YMS base platform [13, p. 32].
- *Multiple selection:* Multiple options exist and arbitrary subsets can be selected. This requires on an implementation level that also the integration / combination of these parts is addressed or is possible. This is in particular relevant for instantiating the Warehouse Management System (WMS) topology, such as the configuration of individual racks [10] or the details about various bins used in a specific warehouse [13, p. 21].
- *Parameterization:* Variation is communicated through a parameter, respectively a value. Variability implementation must support that this value can be referenced in the correct syntax. Examples for such variation in INDENICA is the low-level modification of service configurations, e.g., to realize the service variability in the YMS platform [13, p. 33], the pre-deployment service variability in the Remote Maintenance System (RMS) [13, p. 26] or the specific strategies in the WMS [13, p. 21].
- *Multiplicity:* Repetition of implementation in case that a variability cannot be directly mapped to a parameter, but rather some implementation element must be repeated. This form is, for example, needed to derive initialization scripts for the WMS databases, i.e., multiple data sets derived out of one generic data set describing similar parts of the WMS topology [15].
- *Grouping:* The above operations might be applied to whole groups of elements, such as combining Parameterization and Multiplicity in the case of the WMS database initialization.
- *Extension:* Support the possibility to define that something in the implementation will be augmented by a specific implementation, but it is currently not possible to exactly say what this implementation

might do. One example is to derive optimized implementations of WMS components based on the configured topology in the WMS variability model [10] or instantiated workflows based on a generic workflow artefact and related variability configuration.

Some implementation elements given in this requirement need further information from the specific application context such as the integration / combination in case of multiple selection. This will be addressed below when describing variability instantiation using the concept of production strategies.

RQ2. **Support for artefact-specific instantiation:** Variability instantiation that supports the customization of service platforms must to take into account the various levels of the service platform model introduced in D2.2.1. In particular, the instantiation must support variability in service composition and processes, in domain-specific services, in service and platform deployment and in the technical platform services. This implies that an instantiation must be able to bind variability in different types of artefacts and to cope with possible dependencies among the individual mechanisms used to realize the instantiation. The individual mechanisms may be generic, reusable and configurable (such as a generic template language) or domain- or even platform-specific. We already faced the importance of this differentiation in the INDENICA use cases. On the one hand, the instantiation mechanisms for service configurations in the WMS and RMS case are rather similar, but differ in details [15]. In this case, a reusable instantiation mechanism would be beneficial. On the other hand, the SAP Cocktail approach [11, 15] for the cloud-based YMS is an example for a platform-specific instantiation mechanism.

RQ3. **Support for partial instantiation:** Domain-specific platforms introduce specific needs to variability modelling and instantiation as discussed in D2.2.1. One need is the introduction of domain-specific concepts, which can be easily handled in the variability model as demonstrated in D2.1 for the three INDENICA use cases. As there is no fixed granularity of what a domain is, respectively, any domain may have sub-domains, it is reasonable that platforms can also be initially customized to a broad domain and successively to a more narrowly defined domain. This can be realized by successive (multi-step) and partial instantiation (not only for executable platforms but also for partially configured platforms). Partial instantiation enables pre-configuring the INDENICA base platforms for certain (sub-)domains prior to the integration into a virtual service platform.

RQ4. **Support for cross-sectional variability implementation:** In D2.2.1, several quality characteristics were identified as being important to variability implementation in INDENICA, such as performance requirements, real-time requirements, scalability requirements, reliability requirements or safety requirements. Basically, these characteristics are defined and specified in the variability model. The IVML [10] provides a sophisticated constraint language to express such requirements. In contrast, the variability implementation typically handles quality requirements in terms of varying aspects. Due to their nature as realization of quality requirements, these

aspects are typically rather distributed across an implementation so that variability instantiation in INDENICA must support cross-sectional variability implementation.

RQ5. **Support for binding times:** A binding time defines the latest point in time when the decision about a variability and its implementation must be made. In INDENICA, the support for different binding times is in particular important for the YMS [13, p. 29] and for the RMS [13, p. 26]. This allows to differentiate between variability resolutions, which are made at compile- or deployment-time and fixed afterwards, and dynamic variabilities, which must be resolved at runtime. Examples are the compile-time configuration of the WMS topology or the dynamic adjustment of strategies such as the location services in the YMS or the communication channels in the RMS. The supported binding times as well as their technical details are defined in the variability model. In the IVML [10], binding times are defined in terms of typed attributes, which support individual as well as groups of binding times. Thus, variability instantiation in INDENICA must provide support to relate instantiation mechanisms or combinations of instantiation mechanisms to the generic definitions of binding times as well as their actual values in IVML.

We will realize the given variabilities in terms of production strategies as we will detail in this deliverable. A **production strategy** defines how variant parts must be assembled in the presence of a variability resolution (i.e., a value was assigned to a decision variable). As described in [11], a production strategy is characterized in terms of the definition and evaluation of a variability value, a variation point identification (RQ2, RQ4), techniques for selecting (and combining elements to concretize details left open in RQ1) and a technique for introducing selected elements including relevant glue (RQ2, RQ4). In particular, a production strategy enables the realization of specific forms of variability (RQ1) by (potentially) multiple variability instantiation mechanisms (RQ2) executed at one or multiple binding times (RQ5). Partial instantiation (RQ3) can be realized by successively reapplying production strategies.

In the next section, we will discuss the current support and application of production strategies to the INDENICA case studies.

3 Variability Instantiation in INDENICA

In this section, we discuss the current state of variability instantiation concepts based on the INDENICA case studies as well as their realization in terms of the related tool support developed in INDENICA. We will structure this section into three main parts, the current support for production strategies in WP2 tooling in Section 3.1, the detailed production strategies of the INDENICA case studies realized by the WP2 tooling in Section 3.2 and a summary of the current state in Section 3.3.

3.1 Production Strategies Support

This section details the tool support for production strategies. Basically, the realization of a production strategy needs to identify *what* to transform (the artefacts), *which* configuration values influence the transformation and *how* the actual instantiation shall be performed. In particular, the instantiation described by a production strategy may be realized by one or by multiple instantiation steps, while each instantiation step is realized by an arbitrary number of specific instantiators.

In this section, we will detail the current support for production strategies and instantiators. Therefore, we will first discuss the current architecture for variability instantiation in EASy-Producer, which actually forms a flexible and extensible framework of production strategies and instantiators. Then, we will discuss the configuration of production strategies for application to a specific base platform, and, finally, the specific support for hierarchical product lines (partial instantiation) and service ecosystems.

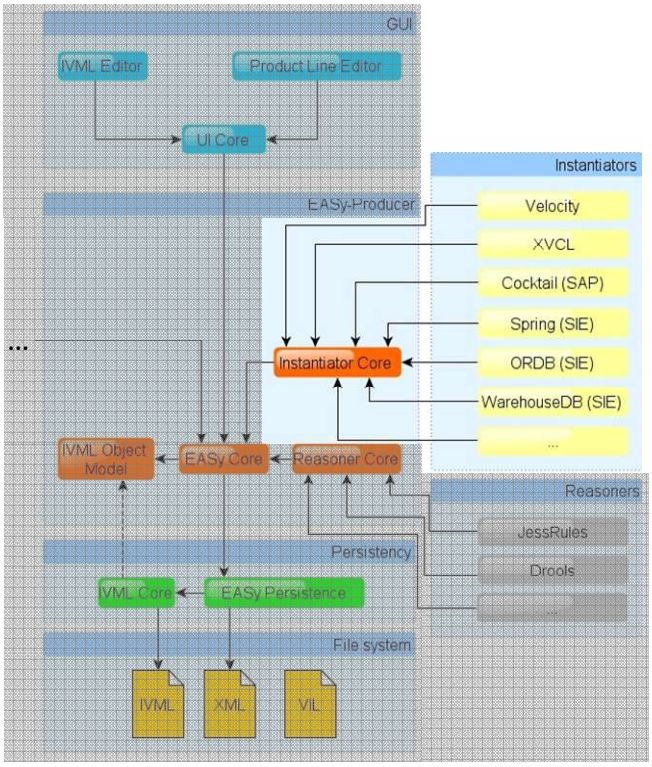

Figure 1: Overview of the EASy-Producer architecture (parts not directly relevant to instantiator-management are greyed out).

3.1.1 EASy-Producer Architecture

EASy-Producer provides an extensible architecture for managing and realizing software product lines. The overall architecture was discussed in detail in D2.4.1. Figure 1 illustrates the overall architecture. In particular, in this deliverable we will focus on the realization of the instantiation part. As shown in Figure 1, in EASy-Producer the instantiation part is realized by the (generic) instantiator core and individual instantiation mechanisms, so called instantiators. We will describe the instantiator core and the individual instantiators in detail in this section.

The design of the variability instantiation support is depicted in Figure 2. The instantiator core component is a general framework for variability instantiation. It realizes the production strategy concept and details it by further related concepts such as artefact management mechanisms or instantiators as we will describe below. The specific mechanisms are either implemented in a reusable way (such as the Java package adaptation or the Velocity instantiator) or realized in terms of case-study specific implementations (such as the Siemens instantiators for the Warehouse Management System). All components are realized as OSGi bundles or in the case of multiple related components as OSGi features in order to simplify the customization of EASy-Producer itself.

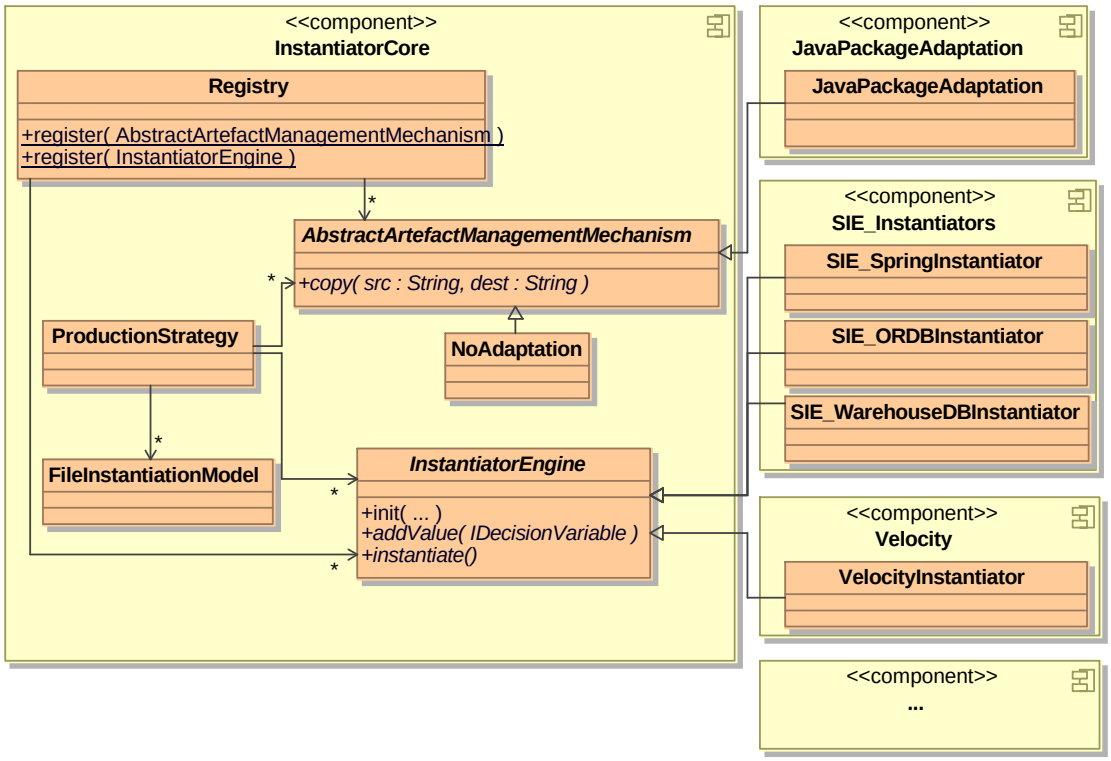

Figure 2: Detailed architecture of the EASy-Producer instantiator core and instantiator components.

The variability instantiation support in EASy-Producer (cf. Figure 2) consists of:

- ***File Instantiation Model***, a collection of artefacts with similar characteristics, i.e., all files which can be instantiated or are already (partially) instantiated in a similar way. These models save also the (link to the) source of every

(partially) instantiated artefact. This facilitates traceability and enables updates if more recent versions of artefacts are available.

- The ***Artefact Management Mechanisms*** are responsible for the preparation of the artefacts which shall be instantiated. This includes provisioning of artefacts, package adaptation to avoid conflicts (an example for software ecosystems is given in Section 3.1.4), file renaming, etc.
- ***Instantiator Engines*** (or short instantiators) perform a specific kind of instantiation on the artefacts provided by an Artefact Management Mechanism. The abstract super class ***InstantiatorEngine*** takes a variability model configuration as input and turns the relevant decision variables and their values into an instantiation context, e.g., to consider only frozen variables during an instantiation.
- A ***Production Strategy*** combines an arbitrary number of File Instantiation Models, Artefact Management Mechanisms, and Instantiator Engines to realize a certain part of the variability instantiation process.
- The ***Registry*** enables the introduction of new Artefact Management Mechanisms and Instantiator Engines to flexibly fulfil the INDENICA variability instantiation requirements. When Eclipse starts, all installed Instantiator Engines and Artefact Management Mechanisms are registered automatically at the Registry through the OSGi Declarative Services mechanism [19].
- The **Instantiation Execution Component** (not shown in Figure 2) handles arbitrary production strategies and is responsible for their execution. In particular, the instantiation execution component handles the execution of different production strategies at different binding times. This component also manages partial instantiation as we will detail in Section 3.1.3.

In summary, the described architecture realizes the requirements for variability instantiation in INDENICA. Individual instantiator engines in combination with a file instantiation model and specific artefact management mechanisms handle the artefact-specific instantiation (RQ2), the specific types of handling variability implementation elements (RQ1), and cross-sectional instantiation (RQ4). Cross-cutting instantiation functionality such as partial instantiation (RQ3) or binding time support (RQ5) is managed by the instantiation execution component.

The initial architecture of EASy-Producer was revised in the context of INDENICA to fulfil the requirements RQ1 – RQ5. Through the plug-in structure, the creation of not yet specified production strategies is supported. Therefore, the developed architecture offers a maximum of flexibility regarding the definition of instantiation processes.

3.1.2 Instantiator Configuration

A production strategy must be configured to enable the instantiation of individual artefacts. As described in Section 3.1.1, a production strategy links instantiator engines, artefact management mechanisms, and file instantiation models to form an instantiation process. This section shows how a new production strategy is currently created in the EASy-Producer user interface.

Usually, the configuration of specific instantiators takes place in the base platform (project) as part of the domain engineering activities, as the platform variant creator [14] of a specific platform may not know which implementation techniques are actually used in the base platform. In some cases, artefacts may also be instantiated from an external source, e.g., from a code repository, so that the related instantiators cannot be configured in the base platform. For this situation, the platform variant creator can configure needed instantiators inside the specific platform. The selection of suitable instantiators for a set of artefacts takes place in the Instantiator View of EASy-Producer. Figure 3 shows the Instantiator View without any configured instantiators.

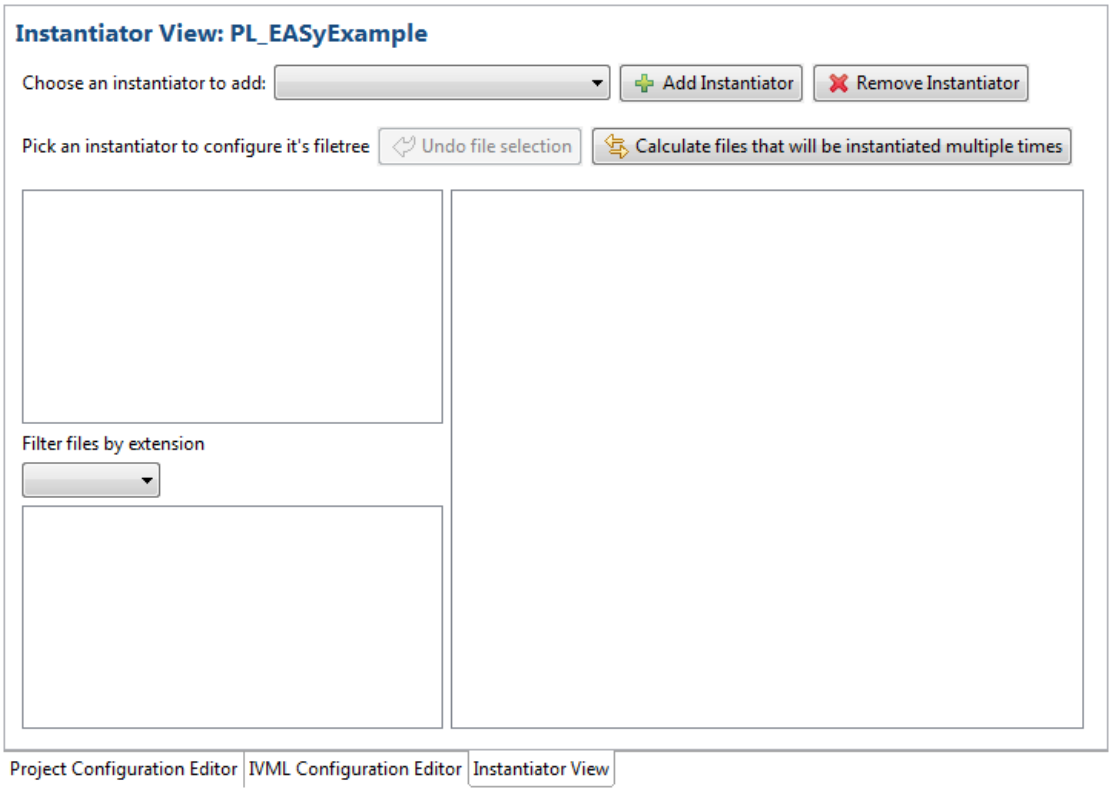

Figure 3: Instantiator View without any configured instantiators.

The drop down menu of Figure 3 shows all installed instantiators, which can be used to specify the current production strategy. Figure 4 shows the selection of such an instantiator. The selected instantiator must be confirmed by pressing the “Add Instantiator” button to add it to the current production strategy. Further instantiators can be added, if needed, e.g., if different types of artefacts are instantiated in specific ways.

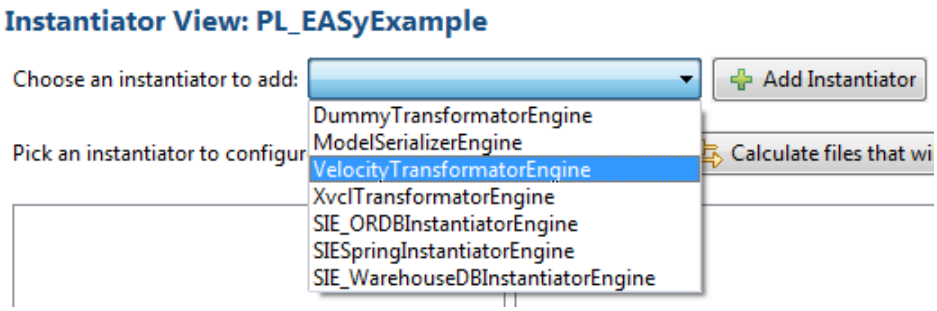

Figure 4: Selection of an Instantiator.

After the instantiators are added to the production strategy, the Instantiator View allows the selection of the artefacts to be applied to the selected instantiators. This step builds up a file instantiator model (cf. Figure 5).

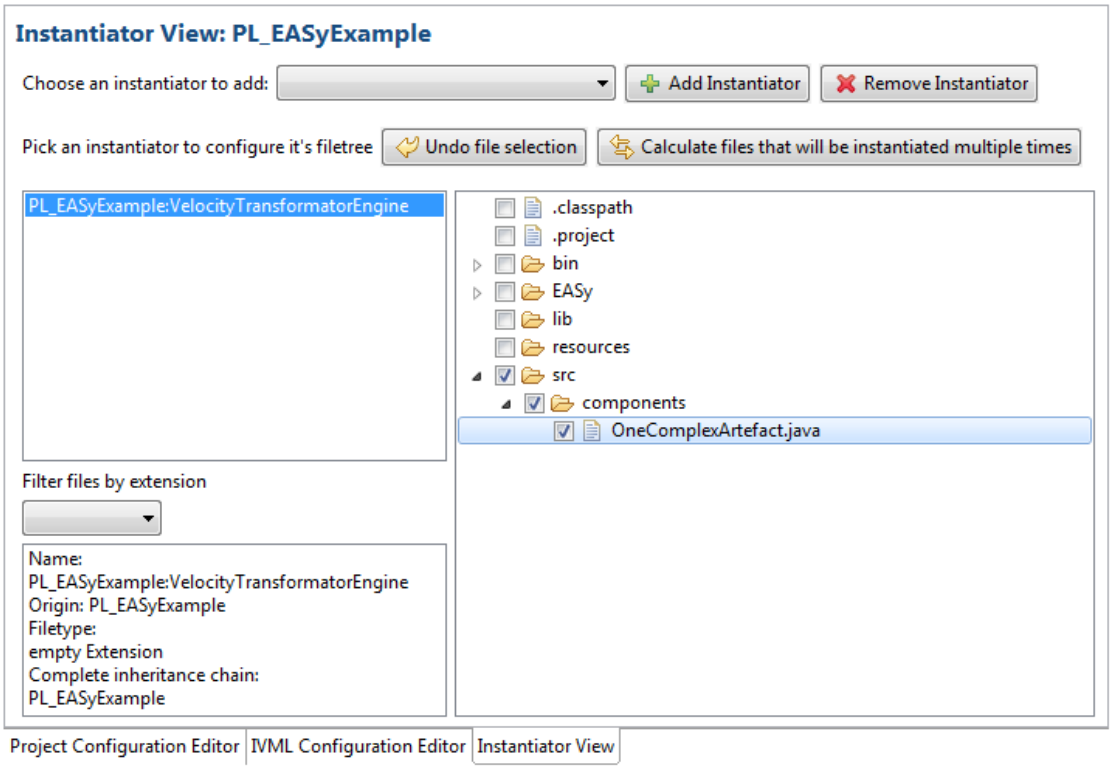

Figure 5: Configuration of files and folders, which shall be instantiated by the currently selected instantiator.

3.1.3 Hierarchical Product Line Support

EASy-Producer supports the management of hierarchical product lines as shown in Figure 6. The example shows a platform, which is partially instantiated for creating localized products on different markets. Such partially instantiated platforms still contain open variability while some variable parts are already resolved.

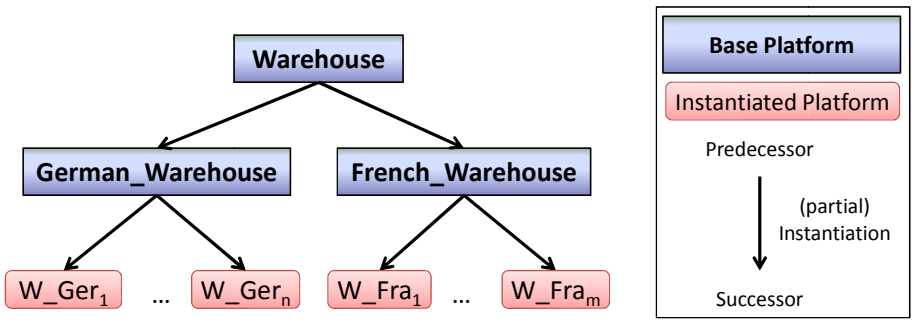

Figure 6: Example of a hierarchical product line.

EASy-Producer does not differentiate between (not instantiated) base platforms and (instantiated) specific platforms for a specific customer to support any combination of (partially instantiated) platforms. Platforms still containing variable parts, i.e., not frozen variables, can be treated as a product line. Such platforms can be used to derive a specific platform to fulfil the customer needs independent of how many instantiation steps were applied before.

In such a hierarchical product line, configured production strategies of a predecessor project will automatically be inherited by all successor projects to ensure a consistent instantiation of all artefacts. Partially instantiated projects can add further sub-production strategies for the instantiation of additional parts, e.g., a variable component, which is only developed for the German market.

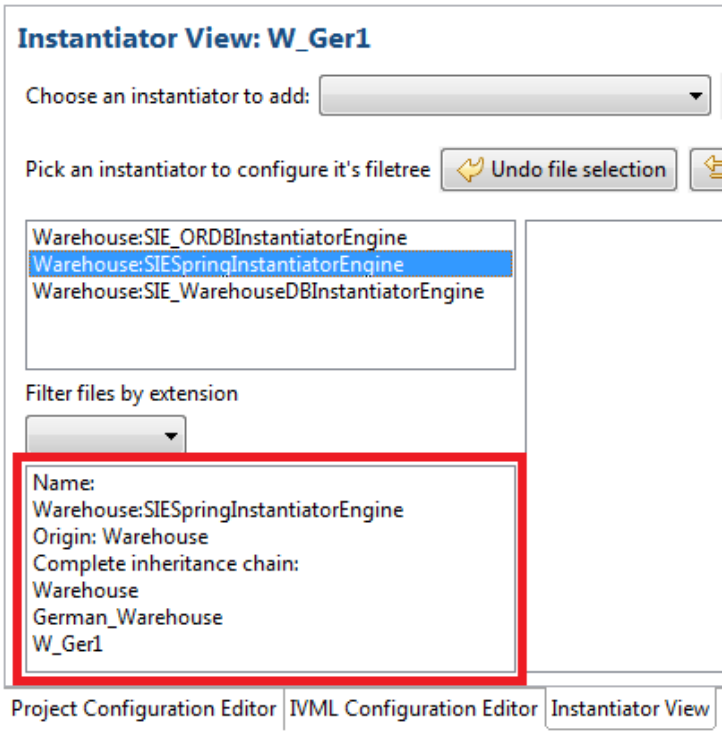

Figure 7: Inherited production strategy of example Figure 6.

Figure 7 shows a Production Strategy of the specific platform "W_Ger_1" of the example in Figure 6. The text area in the lower left corner offers information about the origin of the selected production strategy. The selected production strategy was

created in project “Warehouse” and inherited over “German_Warehouse” to the project “W_Ger$_1$”.

3.1.4 Software Ecosystems Support

Integration of independent service platforms is one of the key objectives of the INDENICA project. For this purpose, an instantiation process is needed which is capable of combining independent base platforms to form an integrated solution. Such an integrated solution can be managed in one project, which simplifies the configuration, instantiation, and extension of the existing assets. This approach is sometimes referred as Multi Software Product Lines (MSPL) [21, 7] and can be applied to service platforms in order to build software ecosystems.

In the context of INDENICA, EASy-Producer is used to realize the configuration of such software ecosystems. One possible MSPL structure is illustrated in Figure 8. The example illustrates how the case studies of Deliverable D5.1 [13] could be combined to a new integrated service platform “Complete_Solution”. This integrated platform could be used to customize all needed service components. Further service components can be added to the platform to integrate the individual parts, i.e., pass data from one platform to another.

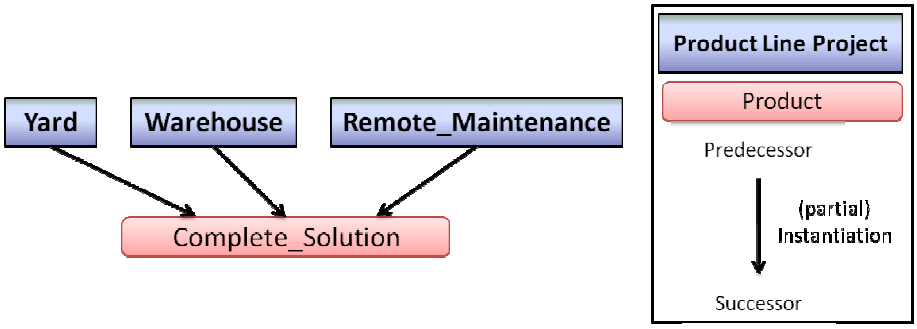

Figure 8: Composition of three software product lines to develop an integrated solution.

EASy-Producer can handle any number of predecessors for a certain platform to facilitate the customization and instantiation of a complete software ecosystem in one step. After the combination of predecessors, a new production strategy is formed consisting of the individual production strategies of the predecessors. Figure 9 demonstrates how the production strategy of “Complete_Solution” from Figure 8 could be assembled in EASy-Producer. The shown production strategy uses five different instantiators to instantiate the individual parts of the three predecessor projects. For each artefact, the correct instantiator will be selected for the instantiation based on the individual file instantiator models. However, instantiation and combining artefacts from different sources may lead to conflicts, such as file name conflicts. Such conflicts can be avoided by applying an appropriate artefact management mechanism, such as the Java package adaptation mechanism shown in Figure 2. This specific mechanism renames entire package hierarchies consistently (including contained classes).

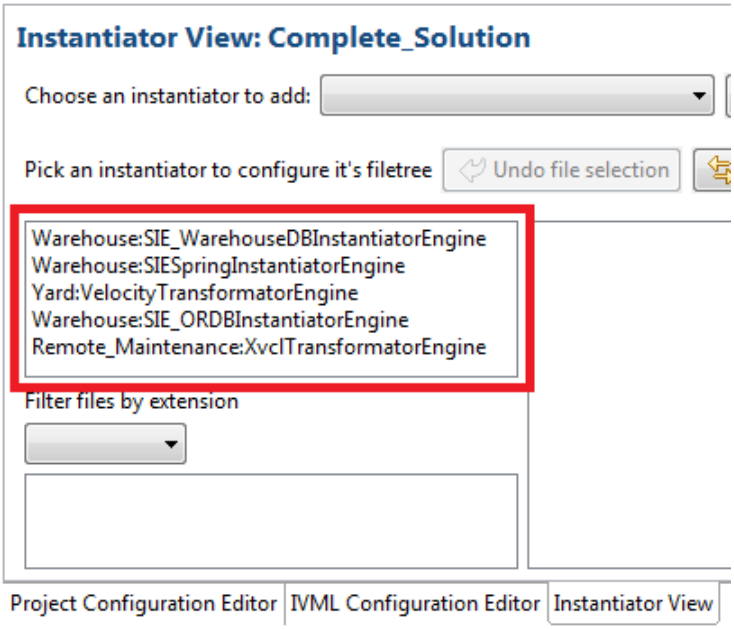

Figure 9: Production strategies of the software ecosystem shown in Figure 8.

As described in Section 3.1.3, EASy-Producer does not differentiate between instantiated and non-instantiated platforms. Therefore, the integrated platform "Complete_solution" could also be used to derive different individual platforms (c.f. Figure 10). Also in such a derivation chain, the correct instantiators selected in the origin projects ("Yard", "Warehouse", and "Remote_Maintenance") would be used to instantiate the assets for the final platforms ("Prod_1", Prod_2", and "Prod_3").

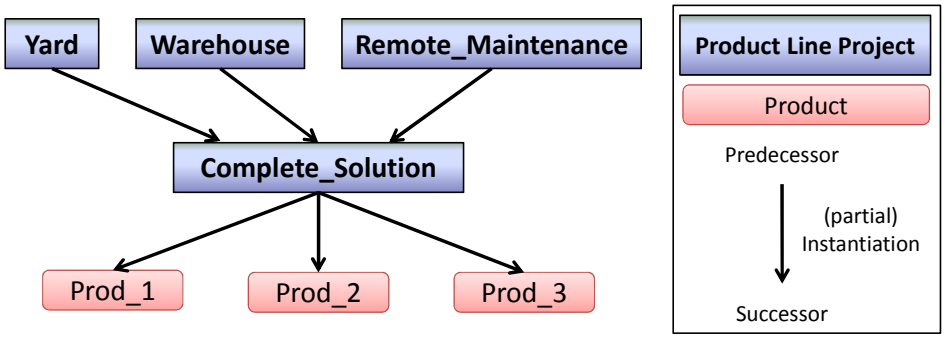

Figure 10: Hierarchical software ecosystem.

3.2 INDENICA Case Studies

In this section, we discuss the individual production strategies, which are currently applied in the INDENICA case studies as well as their realization from a technical point of view. We structure this section according to the case studies and for each case study we will discuss the specific requirements, the production strategy that was developed, the instantiators realizing the production strategy, and examples for artefact instantiations in the respective context. The examples in this section will rely on fragments taken from the individual use cases, such as fragments from the variability model as also discussed in [10] and fragments from specific artefacts. We refrain from describing entire artefacts here in order to focus on the most relevant aspects.

We will describe the individual production strategy in the form of the tables used in Section 5.2 of Deliverable 2.2.1 [11]. In this deliverable we will use one table for an individual production strategy rather than summarized tables as done in D2.2.1. Further, we will also discuss the individual instantiators used to realize the production strategy in terms of an algorithmic description in pseudo code. For this pseudo code, we use the following notation:

- **Bold** denotes typical commands such as foreach, if, etc.
- *Italics* highlights variables, i.e. temporary results
- CAPITAL refers to input or output of an instantiator
- «*TABLE*»-entry matches generic entries in a structured artefact, such as table creations in SQL scripts or element in XML files. «*TABLE*» implicitly defines a temporary variable.
- "String" denotes a constant string
- // marks comments

Typically, an algorithmic description consists of two parts, namely the selection of relevant decision variables and their associated configuration from the variability model and the instantiator algorithm itself. At first glance, the selection step may appear to be superfluous in some instantiators. Basically, the selection step ensures that only frozen variables are used during instantiation. Frozen variables are explicitly fixed at some point in the configuration process so that these values must not change in subsequent partial (staged) configurations. Moreover, the selection step may focus on relevant variables, i.e., a specific subset, instead of working on all available variables. In particular, this is used in the Warehouse Management case in Section 3.2.2.

3.2.1 Yard Management System

In the YMS case study, two different systems are affected by the variability instantiation:

1. The YMS platform itself is customized and instantiated in order to reflect the actual configuration and to enable the modification of selected values at runtime.
2. A web-based user interface is generated which connects to the actual running instance of the YMS and enables the convenient manipulation of the actual values.

The requirements for variability instantiation in the YMS case study are

- Generation of configuration files for the SAP Cocktail mechanism for different binding times [15].
- Execution of the Cocktail mechanism to realize the variabilities in the YMS and to generate the web-based user interface.
- Support of an OSGi-based implementation of the YMS, in particular, OSGi-bundles realized in individual Eclipse projects.

Description of the YMS Production Strategies

The requirements defined above yield two production strategies, one for resolving compile-time and one for runtime variabilities. The compile-time production strategy is summarized in Table 1, while the runtime production strategy is shown in Table 2.

Production strategy element	**Yard management production strategy (Compile-time)**
Definition and evaluation of a variability value	Source code annotations mark the variable attributes and link to the variability model. The specific constant value of an annotated attribute is replaced with the respective value in the configuration.
Variation point identification	Source code annotations
Technique for selecting (and combining) elements	The variability model defines valid binding times per variability (meta-variability). The configuration defines the binding time to be applied as a one-out-of-many selection and, thus, the instantiator to be applied. Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	The Cocktail byte code analyzer searches for the source code annotations and inserts constant values.

Table 1: Cocktail-based compile-time production strategy for the YMS case study.

Production strategy element	**Yard management production strategy (runtime)**
Definition and evaluation of a variability value	Source code annotations mark the variable attributes and link to the variability model. The specific value of an annotated attribute is replaced with the respective value in the (Web-based) configuration through additional code. The additional code enables the manipulation of the actual value entered into the web-based configuration interface.
Variation point identification	Source code annotations
Technique for selecting (and combining) elements	The variability model defines valid binding times per variability (meta-variability). The configuration defines the binding time to be applied as a one-out-of-many selection and, thus, the instantiator to be applied. Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	The Cocktail byte code analyzer searches for the source code annotations and inserts additional code (runtime).

Table 2: Cocktail-based runtime production strategy for the YMS case study.

Cocktail Instantiator

The two production strategies for the WMS case study are realized by a specific instantiator, the Cocktail Instantiator. The instantiator is executed once for a given binding time and, thus, realizes both production strategies at once.

The idea of the Cocktail instantiator is to reuse existing Cocktail tools for realizing variability. This is a well-known concept also in commercial product line tools such as pure::variants [20] or Gears [1], namely using existing tools as a black box.

The algorithmic description of the Cocktail instantiator is given in Figure 11. Basically, the instantiator transforms the variability configuration into a Cocktail specific format and executes Cocktail.

Cocktail Instantiator

Input:

- IVML-configuration IVML_CONFIG //read in from the variability model

Output:

- Cocktail-specific variability model COCKTAIL_VAR_MODEL
- Cocktail-specific resolution COCKTAIL_RESOLUTION

Process:

For Each frozen *variable* **in** IVML_CONFIG
New Cocktail-Variable *var*
var.declaration.name = *variable*.name
var.value = *variable*.value
COCKTAIL_VAR_MODEL.add(*var.*declaration)
COCKTAIL_RESOLUTION.add(*var.*value*)*
End For Each
Execute byte code manipulation on COCKTAIL_RESOLUTION

Figure 11: Algorithmic description of the Cocktail instantiator.

Cocktail Instantiator Example

We will now discuss an example for applying the YMS production strategy. The variability model of the YMS platform contains the following two decision variables shown in Figure 12:

```
String target;
Boolean public;
```

Figure 12: Fragment from the YMS variability model.

Let us assume that the configuration of the YMS platform defines the specific values depicted in Figure 13:

```
target = "indenica.sap.com";
public = true;
```

Figure 13: Fragment from a YMS configuration.

For discussing the instantiation, we will consider the source code artefact from the YMS case study depicted in Figure 14.

```
1  package platform;
2
3  public class Platform {
4
5      @VariationPoint("target")
6      private String deploymentTarget;
7
8      @VariationPoint("public")
9      private boolean isPublic;
10
```

Figure 14: Annotated source code fragment from the YMS case study.

The Cocktail instantiator transforms the YMS variability model into a (simplified[1]) XML representation. Cocktail then takes the XML representation of the variability model as input and manipulates the byte code or generates the Web-based user interface, respectively.

For the compile-time production strategy shown in Table 1, Cocktail produces for compile-time variabilities the code displayed in Figure 15 (instead of byte code we display equivalent source code for convenience). As a result, in Figure 15, the annotations are removed and the specific values from the variability model are inserted as constants in line 5 and line 7.

```
1  package platform;
2
3  public class Platform {
4
5      private String deploymentTarget = "indenica.sap.com";
6
7      private boolean isPublic = true;
8
```

Figure 15: Cocktail output for compile time variability.

For the runtime production strategy given in Table 2, Cocktail inserts additional code for the runtime variabilities, which links the variable attributes to the implementation of the configuration user interface. This is illustrated in Figure 16. Also, here, the annotations are removed (line 5 and 7) as no re-instantiation at runtime is intended. Further, a new code block is inserted (line 9-20). This specific code block links the Java attributes to the runtime mechanism of Cocktail, i.e., the changes in the web-based user interface are reflected in the related attributes.

[1] Cocktail is a code-centric lightweight approach to variability implementation. Thus, its model is much simpler than IVML as it is not intended for full-fledged variability modeling.

```
1  package platform;
2
3  public class Platform {
4
5      private String deploymentTarget;
6
7      private boolean isPublic;
8
9      {
10         cocktail.Cocktail.register("target", new cocktail.Modifier() {
11             public void modify(Object value) {
12                 deploymentTarget = value.toString();
13             }
14         });
15         cocktail.Cocktail.register("public", new cocktail.Modifier() {
16             public void modify(Object value) {
17                 isPublic = Boolean.valueOf(value.toString());
18             }
19         });
20     }
21
```

Figure 16: Cocktail output for runtime variability.

3.2.2 Warehouse Management System

In the WMS case study, four different types of artefacts must be instantiated. One artefact is related to the configuration of actual services in the WMS platform, the remaining three artefacts are related to the database structure and the WMS topology. This leads to the following specific requirements:

- Modification of (parts of) Spring service configuration files.
- Modification of the database schema from a generic artefact using negative variability.
- Modification of the related object-relational database from a generic artefact using variability.
- Generation of a database initialization script, which reflects the WMS topology.
- The variability instantiation must support the implementation of the WMS platform in terms of C# services realized as a Visual Studio Solution [17]. Further, mechanisms provided by the Spring framework [24] and C# related tools must be exploited for variability instantiation.

Description of the WMS Production Strategies

The requirements defined above yield the production strategies summarized in Table 3 to Table 6.

Production strategy element	Warehouse management production strategy (Spring configuration)
Definition and evaluation of a variability value	Service names and individual service parameters are taken over into the Spring configuration file. The mapping is supported by a specific naming convention for the decision variables.
Variation point identification	Markup-elements in the Spring configuration file identify the variation points.
Technique for selecting (and combining) elements	The variability model defines the service configuration. The specific configuration defines the values to be taken over without the need for adjusting the values to the syntax of the target artefact (parameterization). Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	Not needed as generated artefacts are interpreted by the platform implementation.

Table 3: Spring configuration production strategy for the WMS case study.

Production strategy element	Warehouse management production strategy (Database schema)
Definition and evaluation of a variability value	Boolean decision variables configure the presence of specific columns in the database schema. The mapping is supported by a specific naming convention for the decision variables.
Variation point identification	Column names in the SQL scripts identify the variation points.
Technique for selecting (and combining) elements	The variability model defines the database schema configuration. The specific configuration defines the values to be taken over without the need for adjusting the values to the syntax of the target artefact (parameterization). Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	Not needed as generated artefacts are interpreted by the platform implementation.

Table 4: Database schema production strategy for the WMS case study.

Production strategy element	Warehouse management production strategy (Object-relational mapper configuration)
Definition and evaluation of a variability value	Boolean decision variables enable or disable parts of the object-relational mapping configuration. This is supported by a specific naming convention for the decision variables.
Variation point identification	XML elements and related attributes in the object-relational database mapper configuration point to the affected database tables and columns.
Technique for selecting (and combining) elements	The variability model defines the object-relational mapping configuration. The specific configuration defines the values to be taken over without the need for adjusting the values to the syntax of the target artefact (parameterization). Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	Not needed as generated artefacts are interpreted by the platform implementation.

Table 5: Object-relational mapper production strategy for the WMS case study.

Production strategy element	Warehouse management production strategy (Database initialization)
Definition and evaluation of a variability value	The database content reflects the topology of the warehouse, i.e., individual values of structures defined in the variability model are taken over by instantiating templates in the generic artefact. Due to the nature of the warehouse topology, this implies multiplicity as variability implementation type.
Variation point identification	Column names in the SQL scripts identify the variation points.
Technique for selecting (and combining) elements	The variability model defines the database initilization configuration. The specific configuration defines the values to be taken over without the need for adjusting the values to the syntax of the target artefact (parameterization). Multiple selection, i.e., of material entities, is handled by multiplicity as variability implementation type.
Technique for introducing selected elements (including relevant glue)	Not needed as generated artefacts are interpreted by the platform implementation.

Table 6: Database initilization production strategy for the WMS case study.

The WMS production strategies are realized by four specific instantiators. Different instantiators are needed due to artefact-based configuration of instantiators in EASy-Producer.

The specific instantiators are the

1. Spring Configuration File Instantiator.
2. Database Schema Instantiator.
3. Object Relational Mapper Configuration Instantiator.
4. Database Content Instantiator.

The instantiators are executed in the sequence given above. We will detail the individual instantiators in terms of algorithmic descriptions and examples below. However, due to the different artefacts the instantiators are applied to, also the nature of the examples will differ. The example for the Spring configuration instantiator will independently focus on exchanging a service, while the examples for the database instantiators will be related: the example for database schema instantiator will introduce the context of the example and demonstrate how to adjust the database schema of the WMS. Accordingly, the example for the relational

Spring Configuration Instantiator

Input:

- IVML-configuration IVML_CONFIG //read in from the variability model
- List<File> FILES //Spring configuration files to instantiate

Output:

- Instantiated files //all *.config-files in FILES with specific values from the variability configuration

Process:

```
List<DecisionVariable> spring_context
For Each frozen variable in IVML_CONFIG
    add variable to spring_context
End For Each
For Each file in FILES
    For Each «easy:varprop»-Entry in «element»-Entry in «spring»-Entry in
    file
        If «element».name equals “property”
            String varName = «easy:varprop».getContent()
            Decision dec = spring_content.get(varName)
            If dec != null
                «element».setAttribute(“value”, dec.value)
                «easy:varprop».delete()
            End If
        End If
    End For Each
End For Each
```

Figure 17: Algorithmic description of the WMS Spring configuration instantiator.

mapper will reconfigure the mapping of the data to WMS classes. Finally, the example for the database content instantiator will show how the reconfigured data schema impacts the initial creation of the database contents, i.e., the WMS topology.

Spring Configuration File Instantiator

Spring configurations are specified in XML format. The idea for this instantiator is to take an existing Spring configuration file as a template and to modify marked elements, either by inserting a specific configuration value or by removing marked parts based on a certain configuration value (negative variability). Additional XML elements defined according to the Spring specification mark the XML elements to be affected. Thus, a variable Spring specification is again a valid Spring configuration.

Figure 17 shows the algorithmic description of the Spring instantiator for the WMS case study. As the implementation of the WMS platform relies on multiple Spring configuration files, the Spring instantiator may instantiate them as part of a single execution. Basically, the instantiator takes all frozen decision variables into account and tries to match them with the (additional) XML elements in the Spring configuration file. Thereby, the specified value from the variability configuration is taken over and replaced in the marked configuration element. The XML marker is removed after the actual value is inserted into the artefact.

```
project PL_WMS_Platform {

  version v0;

  // further type definitions

  enum StorageAreaType {AMW, MPW, APW, FW, ACW, Other};
  enum ABCStrategy {Abc, Simple};

  // further type definitions

  compound StorageArea {
    StorageAreaType stAreaType;
    ABCStrategy stAreaAbcStrategy;
    Integer BinDepth;
    Boolean VariableBinSizes;
    Boolean MonoMaterial;
    Boolean ColocOpt;

    // further decision variable declarations
  }

  // further type definitions

  StorageArea stArea;

  // further variable declarations

}
```

Figure 18: Fragment from the WMS variability model related to the Spring instantiator.

Spring Configuration File Instantiator Example

We will now discuss the application of the Spring instantiator in terms of an example. Figure 18 depicts the related fragment from the WMS variability model. Type declarations and decision variables not related to the Spring instantiation are not shown in Figure 18.

Let us assume that a WMS configuration defines the variability resolution for the storage area compound depicted in Figure 19 (only the relevant decision variable is shown).

```
stArea = StorageArea {
  stAreaAbcStrategy=ABCStrategy.Abc
  // ...
};
```

Figure 19: Fragment from a WMS configuration related to the Spring instantiator.

We will use the fragment from a WMS Spring configuration file shown in Figure 20 to demonstrate the application of the Spring configuration instantiator. The fragment depicts the service configuration for the alternative storage bin search strategy. Further service configurations and other parts of the Spring configuration file are not depicted in Figure 20. In Figure 20, lines 77-83 contain the marker for the instantiator given in terms of a regular Spring extension (`easy:varprop`). This marker refers to the storage area compound (`stArea`) in the WMS variability model.

```
64  <!-- header and further configuration sections left out here -->
65
66  <spring>
67    <!-- parsers and context left out -->
68
69    <objects xmlns="http://www.springframework.net">
70      <!-- Service definition -->
71      <object id="StorageBinSearchService.StorageBinSearchService"
72        type="StorageBinSearchService.StorageBinSearchService,
73          StorageBinSearchService"
74        singleton="false">
75        <!-- Admissible Values: "Simple", "Abc"-->
76        <property name="Strategy" value="Simple">
77          <easy:varprop
78            xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
79            xsi:schemaLocation=
80              "http://projects.sse.uni-hildesheim.de/easy/spring.xsd"
81            xmlns:easy="http://projects.sse.uni-hildesheim.de/easy/">
82            stArea.stAreaAbcStrategy
83          </easy:varprop>
84        </property>
85      </object>
86    </objects>
87
88    <!-- further sections below left out here -->
89
90  </spring>
```

Figure 20: Fragment from a generic Spring configuration in the WMS case.

The instantiator processes the Spring configuration file and searches for such markers. As `stArea.stAreaAbcStrategy` points to the decision variable determining the storage bin search strategy, the instantiator replaces the value of the parent element of the marker (`easy:varprop`) with the value of this decision variable (here Abc as shown in the configuration above). The result is depicted in Figure 21.

```
64  <!-- header and further configuration sections left out here -->
65
66  <spring>
67    <!-- parsers and context left out -->
68
69    <objects xmlns="http://www.springframework.net">
70      <!-- Service definition -->
71      <object id="StorageBinSearchService.StorageBinSearchService"
72        type="StorageBinSearchService.StorageBinSearchService
73          , StorageBinSearchService"
74        singleton="false">
75        <!-- Admissible Values: "Simple", "Abc"-->
76        <property name="Strategy" value="Abc">
77        </property>
78      </object>
79    </objects>
80
81    <!-- further sections below left out here -->
82
83  </spring>
```

Figure 21: Instantiated fragment from the WMS Spring configuration shown in Figure 20.

Database Schema Instantiator

The database schema instantiator is responsible for adjusting the database schema of the WMS according to the WMS variability configuration.

Basically, the database schema instantiator realizes negative variability, i.e., the underlying SQL artefact contains the code for the case that all variabilities are enabled. However, not all decision variables in the WMS IVML model shall affect the database schema, such as those related to the WMS topology as they represent data rather than the underlying schema. A simple mechanism to realize this separation is a naming schema, i.e., this instantiator considers only decision variables, which end with “Config”. The instantiator will remove the disabled variabilities. The result will contain a subset of the SQL statements of the input file. The algorithmic description of the database schema instantiator is shown in Figure 22.

Database Schema Instantiator

Input:

- IVML-configuration IVML_CONFIG //read in from the variability model
- Single SQL-file SQL_FILE //SQL file to instantiate

Output:

- SQL_FILE with adapted/customized table columns

Process:

```
List<DecisionVariable> db_context
For Each frozen variable in IVML_CONFIG
with variable.name ends with "Config"
        add variable to db_context
End for Each
For Each «CREATE TABLE»-entry in SQL_FILE
  For Each variable in db_data_context
    If «CREATE TABLE ».name equals variable.compoundName
      For Each «column» -entry in «CREATE TABLE»-entry
        If "db"+ «column».name +"Config" equals variable.name
        && variable.value == false
          Delete « column»-entry
        End If
      End For Each
    End If
  End For Each
End For Each
```

Figure 22: Database Schema Instantiator for the WMS case study.

Database Schema Instantiator Example

The example for this instantiator focuses at changing the database schema and, thus, will define the data structures to which the remaining two WMS database instantiators must comply. Therefore, we will introduce a common example here and continue this example in the discussion of the remaining two instantiators.

The WMS variability model contains compounds for configuring the data managed by the WMS. One example is the compound `MaterialConfig` shown in Figure 23, which describes the (additional information of the) material to be stored in a warehouse[2].

[2] An example involving the rows, columns and racks of a warehouse would increase the space needed to describe the example. As the same mechanisms are applied to these parts of the topology configuration, we will discuss the simpler case here for convenience.

```
compound MaterialConfig {
    Integer materialNumber;
    String name;
    Boolean description;
    // further contained decision variables
}
```

Figure 23: Fragment from the WMS variability model related to the database configuration.

Currently, the name of the compound (variable) indicates whether a compound may impact the underlying WMS database structure. A more sophisticated specification of the instantiation description language would provide details in terms of a mapping specification rather than a naming convention.

We now show how the WMS variability model can be used to configure the data managed by the WMS. As an example, we disable the descriptive information for the material as shown in Figure 24:

```
materialConfig = MaterialConfig {
  description = false
  // configuration of further variables
};
```

Figure 24: Fragment from a WMS configuration related to the database configuration.

Figure 25 depicts a fragment from the WMS database creation script, here the part for creating the material table in terms of SQL statements. Basically, it creates the individual data structures representing the WMS topology in terms of database tables and data columns.

```
/** fragment: table creations above such as rack skipped **/

/****** Object:  Table [dbo].[material]    Script Date: 06/11/2012 13:52:26 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[material](
    [materialNumber] [nvarchar](50) NOT NULL,
    [name] [nvarchar](50) NULL,
    [description] [nvarchar](50) NULL,
PRIMARY KEY CLUSTERED
(
    [materialNumber] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
      ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO

/** fragment: table creations below such as locationtype skipped **/
```

Figure 25: Fragment from the WMS database creation SQL script.

Applying the database schema instantiator, the fragment in Figure 25 will be modified as shown in Figure 26. As a result, columns related to the disabled variability will be removed from the initialization script.

```
/** fragment: table creations above such as rack skipped **/

/****** Object:  Table [dbo].[material]    Script Date: 06/11/2012 13:52:26 ******/
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
CREATE TABLE [dbo].[material](
    [materialNumber] [nvarchar](50) NOT NULL,
    [name] [nvarchar](50) NULL,
PRIMARY KEY CLUSTERED
(
    [materialNumber] ASC
)WITH (PAD_INDEX  = OFF, STATISTICS_NORECOMPUTE  = OFF, IGNORE_DUP_KEY = OFF,
      ALLOW_ROW_LOCKS  = ON, ALLOW_PAGE_LOCKS  = ON) ON [PRIMARY]
) ON [PRIMARY]
GO

/** fragment: table creations below such as locationtype skipped **/
```

Figure 26: Instantiated fragment from Figure 25.

Object-relational Mapper Configuration Instantiator

An object-relational mapper is responsible for providing a program with uniform access to the database contents in form of Database Access Objects (DAO). Dependent on the specific mapper, advanced functionality such as caching or transactions may be handled by the DAOs or the mapper in a transparent fashion. Further, access to the data is simplified as error prone manual access statements are generated and encapsulated in the DAOs. In the specific case of the WMS, the object-relational mapper of Visual Studio is also responsible for the mapping of database tables and columns to user interface components. Consequently, modifying the mapper configuration in a consistent way enables to configure the managed data as well as their access from the user interface. In summary, the configuration for the object-relational mapper in the WMS platform consists of four parts, namely,

- The *storage model* describes the database.
- The *conceptual model* describes the C#-classes to be generated.
- The *data mapping* describes the mapping between database and C#-classes as well as their properties
- The *graphical mapping* to user interface elements.

Object-Relational Mapper Instantiator

Input:

- IVML-configuration IVML_CONFIG //read in from the variability model
- Single edmx-file MODEL_FILE //generic mapper configuration artefact

Output:

- Customized MODEL_FILE //instantiated mapper configuration

Process:

```
List<DecisionVariable> or_mapper_context
For Each frozen variable in IVML_CONFIG
with variable.name ends with "Config"
        Add variable to or_mapper_context
End For Each
For Each sub_model in MODEL_FILE
   For Each variable in or_mapper_context
      For Each «EntityType»-entry
      with «EntityType».Name + "Config" equals variable.name
         For Each «Property»-entry
            If variable.hasVariable(«Property».name)
            && variable.getVariable(«Property».name).value == false
               Delete «Property»-entry
            End If
         End For Each
      End If
   // similar for ScalarPropert in MappingFragment, AssociationSet, etc.
   End For Each
End For Each
```

Figure 27: Instantiator for the object-relational mapping in the WMS case study.

The idea for this instantiator is rather similar to the database schema instantiator discussed above. The object-relational mapper instantiator realizes negative variability, i.e., a configuration file for all enabled variabilities is used as a generic artefact and disabled variabilities are removed. Following the same naming schema as the database schema instantiator, the object-relational mapper instantiator considers only decision variables, which end with "Config". The algorithmic description of the instantiator is given in Figure 27.

Object-relational Mapper Configuration Instantiator Example

In this example, we will rely on the variability model fragment discussed above for the database schema instantiator, i.e., the `MaterialConfig`. We will apply the same configuration as shown above, i.e., the descriptive text is disabled.

Figure 28 depicts a fragment of the object-relational mapper configuration of the WMS having all variabilities enabled. In this fragment the definitions of entity sets, association sets and the user interface designer section are not shown as they are not relevant for this example.

```
2  <edmx:Edmx Version="2.0" xmlns:edmx="http://schemas.microsoft.com/ado/2008/10/edmx">
3    <!-- EF Runtime content -->
4    <edmx:Runtime>
5      <!-- SSDL content -->
6      <edmx:StorageModels>
7        <Schema Namespace="wmsdbModel.Store" Alias="Self" Provider="System.Data.SqlClient"
8          ProviderManifestToken="2008"
9          xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
10         xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
11         <!-- definitions and association sets skipped -->
12         <EntityType Name="material">
13           <Key>
14             <PropertyRef Name="materialNumber" />
15           </Key>
16           <Property Name="materialNumber" Type="nvarchar" Nullable="false" MaxLength="50" />
17           <Property Name="name" Type="nvarchar" MaxLength="50" />
18           <Property Name="description" Type="nvarchar" MaxLength="50" />
19         </EntityType>
20         <!-- further entity types skipped -->
21       </Schema></edmx:StorageModels>
22       <!-- conceptual models and mappings skipped -->
23   </edmx:Runtime>
24   <!-- designer content skipped -->
25 </edmx:Edmx>
```

Figure 28: Fragment of the generic database mapper configuration in the WMS case.

Given the WMS variability model and the specific configuration discussed above, the generic mapper configuration in Figure 28 will be instantiated as shown in Figure 29. Thereby, in particular the property in line 18 is removed. Related properties in the conceptual models and mappings sections are handled similarly and, thus, they are not displayed in Figure 28 and Figure 29. Please note that also cross-relations such as association sets, associations or association connectors are affected if required. This is not shown here in order to keep the example and the fragments readable and understandable.

```
<edmx:Edmx Version="2.0" xmlns:edmx="http://schemas.microsoft.com/ado/2008/10/edmx">
  <!-- EF Runtime content -->
  <edmx:Runtime>
    <!-- SSDL content -->
    <edmx:StorageModels>
      <Schema Namespace="wmsdbModel.Store" Alias="Self" Provider="System.Data.SqlClient"
        ProviderManifestToken="2008"
        xmlns:store="http://schemas.microsoft.com/ado/2007/12/edm/EntityStoreSchemaGenerator"
        xmlns="http://schemas.microsoft.com/ado/2009/02/edm/ssdl">
        <!-- definitions and association sets skipped -->
        <EntityType Name="material">
          <Key>
            <PropertyRef Name="materialNumber" />
          </Key>
          <Property Name="materialNumber" Type="nvarchar" Nullable="false" MaxLength="50" />
          <Property Name="name" Type="nvarchar" MaxLength="50" />
        </EntityType>
        <!-- further entity types skipped -->
      </Schema></edmx:StorageModels>
      <!-- conceptual models and mappings skipped -->
  </edmx:Runtime>
  <!-- designer content skipped -->
</edmx:Edmx>
```

Figure 29: Instantiated database mapper script.

Database Content Instantiator

The database content instantiator is the last instantiator executed in sequence as part of the WMS production strategy. The idea of this instantiator is to create a database initialization script from scratch. This is required to produce output which is consistent with the results of the previous instantiators, i.e., disabled fields or relationships must not be instantiated. While this may also be achieved with negative variability, this instantiator must also create an initialization for an arbitrary number of instances, such as a material list or a set of racks. As the database structure is reflected in the variability model, we can easily create the SQL initialization script from scratch. As discussed above, also this instantiator follows the naming convention for decision variables, i.e., the instantiator will consider decision variables related to the WMS topology rather than to the database structure. Moreover, the alignment between database structure and WMS topology already starts in the variability model as we will show below.

The algorithmic description of the database content instantiator is depicted in Figure 30. Basically, it considers all frozen decision variables of the WMS variability model for which the name ends with “Topology” due to the WMS specific naming convention. For these decision variables, it creates the corresponding SQL insert statements and for each defined variable the related SQL data-value assignments. Here, only defined variables are considered, i.e., those which have not been removed from the database schema, and the mapping by the preceding instantiators.

Database Content Instantiator

Input:

- IVML-configuration IVML_CONFIG //read in from the variability model
- OUTPUT_FILE_NAME

Output:

- SQL_FILE with adapted/customized INSERT-statements

Process:

```
List<DecisionVariable> db_data_context
For Each frozen variable in IVML_CONFIG ends with "Topology"
        add variable to db_data_context
End For Each
New SQL-FILE with OUTPUT_FILE_NAME
New «USE»-Entry on "wmsdb"
For Each variable in db_data_context
   For Each element in variable // matches (non-)container contents
      new «INSERT»-Entry in SQL_FILE
      «INSERT».name = substring(variable.name, -"Topology".length)
      For Each var in variable
         If isDefined(var)
            new «COLUMN»-Entry in «INSERT»
            «COLUMN».name = var.name
            «COLUMN».value = var.value
         End If
      End For Each
   End For Each
End For Each
```

Figure 30: Database content instantiator for the WMS use case.

Database Content Instantiator Example

Also in this example, we will rely on the context of the examples discussed above for the database schema instantiator and the object-relational mapper instantiator. After executing these two instantiators, the database schema is defined and correctly mapped to DAOs. The final step in the WMS production strategy is to turn the WMS topology defined in the actual WMS variability configuration into database entries.

The compound `MaterialConfig` discussed above defines the data to be managed in the WMS. In contrast, the compound `Material` shown in Figure 31 is responsible for configuring the material part of the WMS topology, i.e., the specific material to

be inserted into the initial WMS database. As the actual structure of `Material` can be configured by `MaterialConfig`, the instances of `Material` must be aligned with the actual configuration in the decision variable `materialConfig`. This is done by defining a constraint in the WMS variability model as shown in Figure 31. Finally, the container `materialTopology` contains the actual material as part of the WMS topology configuration.

```
compound Material {
  Integer materialNumber;
  String name;
  String description;
  materialConfig.description = isDefined(description);
  // similar for further contained decision variables
}

setOf(Material) materialTopology;
```

Figure 31: Fragment from the WMS variability model related to the WMS topology.

Figure 32 depicts a fragment from a WMS configuration specifying a part of the WMS topology, here the initially available material. The `description` information was disabled in the related examples above and, thus, cannot be configured here:

```
materialTopology = {
  Material{
    materialNumber = 1,
    name="Box with Screws, 15mm"},
  Material{
    materialNumber = 2,
    name="Screwdriver Nr. 4"}
};
```

Figure 32: Fragment from a WMS configuration related to the WMS topology.

Consequently, the database content instantiator shown in Figure 30 will produce the fragment from the database initialization script depicted in Figure 33. The insert statements are generated by the instantiator due to the WMS topology configuration. The specific lines for `materialTopology` are shown in lines 4-5, while the remaining lines of the fragment are generated from other parts of the WMS topology configuration not discussed in the examples above.

```
1  USE [wmsdb]
2  GO
3
4  INSERT [dbo].[material] ([materialNumber], [name]) VALUES (N'1', N'Box with Screws, 15mm')
5  INSERT [dbo].[material] ([materialNumber], [name]) VALUES (N'2', N'Screwdriver Nr. 4')
6
7  /****** configured by other parts of the WMS variability model ******/
8
9  INSERT [dbo].[rack] ([id], [name]) VALUES (N'R', N'Rack')
10
11 INSERT [dbo].[locationtype] ([id], [name]) VALUES (N'1', N'Conveyor')
12 INSERT [dbo].[locationtype] ([id], [name]) VALUES (N'2', N'RackFeeder')
13 INSERT [dbo].[locationtype] ([id], [name]) VALUES (N'3', N'PickingStation')
```

Figure 33: Generated database content initialization script reflecting the configured WMS topology.

3.2.3 Remote Management System

For the RMS case study, also a platform-specific service configuration file must be handled. In addition, the constraints on the runtime variables defined in the variability model shall be made available to the WP4 adaptivity manager in order to support the determination of the actual configuration values at runtime. This leads to the following specific requirements:

- Modification of (parts of) a service configuration file at compile-time.
- Creation of a reasoning-rules artefact which restricts the possible adaptation actions at runtime.
- Variability instantiation in the context of the NDL-specific extension of Mobicents / JBoss environment, i.e., Java projects with dependencies managed by Maven.

Description of the RMS Production Strategies

The requirements defined above yield two distinct production strategies, one for compile-time and one for runtime. The compile-time production strategy is summarized in Table 7. The runtime production strategy is depicted in Table 8. In the RMS case study, these strategies are realized by two specific instantiator, a service configuration instantiator and a rules instantiator.

Production strategy element	**Remote management production strategy (compile time)**
Definition and evaluation of a variability value	Configuration values are directly taken over into the specific artefacts. At compile time, enabled services as well as their specific parameters are taken over into the RMS configuration file.
Variation point identification	Specific markup elements in service configuration file.
Technique for selecting (and combining) elements	Some configuration values can be taken over directly following the syntax and structure of the target artefact. However, for some values also explicit mappings must be specified. Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	Not needed.

Table 7: Compile-time production strategy in the RMS case study.

Production strategy element	Remote management production strategy (runtime)
Definition and evaluation of a variability value	Configuration values are directly taken over into the specific artefacts. The RMS platform provides services for setting the related decision variable values. The values are interpreted by the RMS platform. Constraints referring to runtime variables are written to a rule format for further use in WP4 to support the runtime adaptation of the RMS platform.
Variation point identification	No variation point identification needed for runtime configuration as the result is generated from scratch.
Technique for selecting (and combining) elements	The rules can be directly generated for the target language. Multiple selection is not possible.
Technique for introducing selected elements (including relevant glue)	Not needed.

Table 8: Runtime production strategy in the RMS case study.

Service Configuration Instantiator

The RMS service configuration is defined in terms of an XML file following a platform specific structure. In that structure, pre-deployment (compile-time variables) as well as default values for runtime variables are specified. However, some element names, structures or even configuration values cannot directly be mapped to the variability model as this is the case in the YMS case study (cf. Section 3.2.1) or in the WMS case study (cf. Section 3.2.2). Therefore, the idea for the Service configuration instantiator is similar to the Spring configuration instantiator in Section 3.2.2, i.e., to mark affected XML elements for inserting values. In contrast, the values for the RMS configuration may require a specific mapping.

The algorithmic description of the service configuration instantiator is depicted in Figure 34. The instantiator selects the frozen variables from the IVML configuration and performs a mapping to the XML structure while replacing existing values as well as markup elements by specific values from the configuration. Thereby compile-time variabilities are considered in order to fill the values of the pre-deployment section described above. The instantiator also performs value mapping as stated in Figure 34 in terms of the `map` function. The `map` function either considers the global value mappings bound to a certain IVML type or specific mappings defined in the markup element (attribute `mapping` of the element `easy:varprop`).

RMS Service Configuration Instantiator

Input:

- IVML-configuration IVML_CONFIG //read in from the variability model
- File FILE //RMS configuration file to instantiate

Output:

- Instantiated file

Process:

List<DecisionVariable> *rms_context*
For Each frozen *variable* **in** IVML_CONFIG **with** isDefined(variable)
  add *variable* **to** *rms_context*
End For Each
For Each «*easy:varprop*»-Entry **in** «*element*»-Entry **in** «*indenica-rms-main-config*»-Entry **in** *file*
  For Each *variable* **in** *rms_context*
    If getContents(«easy:varprop») **equals** variable.name
      «*element*».setContents(map(«*easy:varprop*».value, «*easy:varprop*».mapping))
      «*easy:varprop*».delete()
    End If
  End For Each
End For Each

Figure 34: Algorithmic description of the RMS service configuration instantiator.

Service Configuration Instantiator Example

Now, we will discuss the application of the RMS service configuration instantiator. The RMS variability model contains the (compile-time) decision variables as shown in Figure 35. For convenience, we will take the default values of these decision variables as the actual configuration.

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

ServiceState videoCalls = ServiceState.point2point;
ServiceState audioCalls = ServiceState.full;
ServiceState textMessaging = ServiceState.full;
Boolean userPositioning = false;
// further decision variables and constraints
```

Figure 35: Fragment from the compile-time part of the RMS variability model.

A fragment from the generic RMS service configuration file is depicted in Figure 36. Basically, we focus on the specific configuration options shown in the variability model fragment above and leave out similar configuration options.

```
2  <indenica-rms-main-config
3      xmlns:ndl="http://www.nextdaylab.com/indenica/rms/xml">
4      <easy:mapping type="ServiceState"
5        mapping="full=multipoint,point2point=p2p"/>
6      <pre-deployment>
7          <config-version>0.7</config-version>
8          <communication-type>
9              <text>
10               <easy:varprop>textMessaging</easy:varprop>p2p
11             </text>
12             <voice>
13               <easy:varprop>audioCalls</easy:varprop>off
14             </voice>
15             <video>
16               <easy:varprop>videoCalls</easy:varprop>off
17             </video>
18         </communication-type>
19
20         <!-- further setting groups -->
21
22         <user-positioning>
23           <easy:varprop>userPositioning</easy:varprop>true
24         </user-positioning>
25
26         <!-- boolean settings -->
27     </pre-deployment>
28
29     <!-- runtime default values -->
30 </indenica-rms-main-config>
```

Figure 36: Fragment from the RMS configuration file.

As mentioned above, several configuration settings in the configuration file cannot directly be mapped to the decision variables in the variability model regarding structure, naming or even configuration values. For example, the communication type setting for text, voice and video in lines 9-17 does neither match in (nesting) structure nor in element naming to the variability model. In addition, also the configuration values themselves do not map to the types in the variability model, e.g., the value `multipoint` corresponds to `ServiceState.full` or the value `point2point` to `ServiceState.p2p` while `ServiceState.off` can be mapped directly. Therefore, we define a (partial) mapping for the type `ServiceState` in line 10.

The instantiator will produce the instantiated artefact as shown in Figure 37. However, the mapping specification remains in the configuration file in order to support staged instantiation of currently unresolved variabilities. In case that all variabilities are actually resolved, the mapping definitions may be removed by the instantiator.

```
<indenica-rms-main-config
    xmlns:ndl="http://www.nextdaylab.com/indenica/rms/xml">
    <easy:mapping type="ServiceState"
      mapping="full=multipoint,point2point=p2p"/>
    <pre-deployment>
        <config-version>0.7</config-version>
        <communication-type>
            <text>multipoint</text>
            <voice>multipoint</voice>
            <video>p2p</video>
        </communication-type>

        <!-- further setting groups -->

        <user-positioning>
          true
        </user-positioning>

        <!-- boolean settings -->
    </pre-deployment>

    <!-- runtime default values -->
</indenica-rms-main-config>
```

Figure 37: Instantiated service configuration file in the RMS case study.

Runtime Rules Instantiator

Currently, the RMS platform relies on parameterization for runtime variabilities, i.e., specific values are defined by an external mechanism such as the WP4 adaptation manager and passed in using a specific Web service interface. However, the values determined by the adaptation manager must comply with the constraints on runtime variabilities in the variability model. In extreme cases, the constraints may even be used to determine appropriate adaptation values.

The idea for the runtime rules instantiator is to export the constraints from the variability model which characterize runtime variabilities to a text file. In the future, the rules will be exported to the INDENICA infrastructure repository. As the resulting rules shall be used as input to the WP4 adaptivity manager, they must be exported in terms of Drools rules [12]. This can easily be done as the main reasoner of EASy-Producer relies on Drools as described in [15]. Given a specific configuration, the instantiator requests the instantiated pre-runtime decision variables from the Drools reasoner, i.e., it removes resolved variables and fulfilled constraints. Then, the instantiator turns the constraints from the variability model into Drools rules and

stores the rule set into a file for further use. Thus, the algorithmic description for the runtime rules instantiator in Figure 38 is fairly simple.

RMS Runtime Rules Instantiator

Input:

- IVML variability model IVML_MODEL
- IVML-configuration IVML_CONFIG //read in from the variability model
- String FILE_NAME //output file name

Output:

- Instantiated file

Process:

VarModel model = DroolsReasoner.instantiatePriorRuntime
(IVML_MODEL, IVML_CONFIG)
String rules = DroolsReasoner.getOpenRules(IVML-MODEL, model)
New FILE **with** FILE_NAME
FILE.write(*rules*)

Figure 38: Algorithmic description of the RMS rules instantiator.

Runtime Rules Instantiator Example

An excerpt of the rules produced by the RMS rules instantiator for the runtime-related constraints in the RMS variability model is depicted in Figure 39.

```
rule "35"
agenda-group "Group 2"
dialect "mvel"
no-loop true
 when

P0 : Project0()
P : Project(
    eval(
    dList.contains("serversAtHalfFullSpeed")  && isDefined(P0.currentServerState)
    && isDefined(P0.currentServerState) && dList.contains("currentServerState.availableVMs")
    && dList.contains("runAdditionalVMs") &&  !(implies(P0.getServersAtHalfFullSpeed()
    && P0.currentServerState.availableVMs > 0 , P0.getRunAdditionalVMs() > 0 )))
    )
then
cList.add(35);
end
```

Figure 39: Instantiated runtime rules from the RMS variability model.

3.2.4 Additional Case Studies

In addition to the work on variability instantiation in the INDENICA case studies, we validated our instantiation approach in further contexts. The additional case studies provided further insights and enable the integration of the gained experience and realized technologies into the INDENICA case studies. We will briefly discuss the additional case studies below.

- Based on prior work published in [11, 22, 23], we carried out additional experiments on shifting the **binding time** in existing systems (**meta-variability**). This is realized by an instantiator, which generates multiple artefacts for introducing binding-time specific glue code involving template instantiation, aspect-oriented bytecode injection (here used as a pre-processor), and build-process adjustments (if required). This received significant interest. In addition, further experiments on meta-variability of component technologies have been conducted. Currently, this work is integrated into EASy-Producer by applying the INDENICA variability instantiation concepts.
- In a master thesis, concepts and mechanisms for realizing **deployment variability** for cloud-based applications were developed. Thereby, additional instantiators for customizing the deployment mechanisms of cloud applications have been realized and validated using EASy-Producer [8].
- The national funded project **ScaleLog** focuses on scaling intralogistics software for warehouses of different sizes. One particular aim for Klug Integrierte Systeme, the industrial partner in ScaleLog, is to flexibly address new markets of medium- and small-size warehouses while providing a clear migration path to larger warehouses for potential customers. In ScaleLog, further **project-specific instantiators** for EASy-Producer have been developed and, thus, the INDENICA variability instantiation concepts have been validated and applied. Some examples are a variant of the Spring configuration instantiator discussed in Section 3.2.2, a template-based workflow implementation instantiator and an instantiator which customizes the user interface. Currently, a more complex instantiator is being developed, which combines workflow modelling with variability instantiation using code generation.
- **EASyCar** is a bilateral project with the Robert Bosch GmbH in Stuttgart / Feuerbach. The aim of this project is to research modeling concepts for **large-scale variability** in the automotive domain. One step towards this aim is to evaluate variability modeling, variability instantiation and the related tool support provided by EASy-Producer in a business unit at Bosch. Therefore, variability model import mechanisms as well as **further instantiators** have been developed. One particular challenge is to practically apply EASy-Producer to a large-scale variability model, i.e., the IVML editor, the configuration editor, the reasoning support and the instantiation framework.

The experience made in the projects and works described above drives our aim in improving and generalizing the existing INDENICA variability instantiation concepts as we will further outline in Section 4.

3.3 Summary

The realization of production strategies discussed in this section enables the instantiation of variabilities for the individual INDENICA case studies. Additional instantiations provide the instantiation functionality to apply the concepts and tools also in other projects in an extensible way, such as ScaleLog or EasyCar. The WP2 tool support described and demonstrated in this section fulfils the requirements from Section 2. We discussed this fulfilment in Section 3.1 in terms of the realized tool support and in Section 3.2 in terms of the various use cases.

Although this is a good and solid baseline, our practical work on the INDENICA case studies as well as on additional case studies shows that the current realization can be improved. The current realization of the INDENICA variability instantiation concepts can be summarized as Java-based implementation of production strategies. Some particular instantiators may need a specific implementation, such as the (integration of the) Cocktail instantiator described in Section 3.2.1. However, the current implementation-based approach also implies a technical overhead to apply and (partially) reuse existing instantiators. For example, the Spring configuration instantiator in Section 3.2.2, the RMS configuration instantiator in Section 3.2.3 and even the mapping instantiator in Section 3.2.2 share several similarities, in particular as they instantiate XML artefacts.

Based on our practical experience, we learned the following lessons. We will explicitly name the INDENICA instantiators, we draw our lessons from, although we made similar experience in the additional case studies.

E1. Variability instantiation may create entire files (WMS database instantiator) or modify individual fragments within a file (RMS configuration file instantiator).
E2. Currently, individual instantiators translate parts of the variability model into the result artefact. This translation is implemented in Java, but can be abstracted using a template language. Such a template language must be able to take over configuration values (WMS spring instantiator), iterate over structures in the variability model (WMS database instantiator) or map values to artefact values (RMS configuration instantiator). We expect that even simple calculations may be helpful in describing such mappings.
E3. The realization of runtime variability leads to the translation of parts of the information on a runtime variability into another language, such as the linking code created by the YMS Cocktail instantiator. In general, instantiation of runtime variability may happen on specific types of artefacts such as classes or components.
E4. Even a template-based realization of variability instantiation may rely, in special cases, on external tools and mechanisms, i.e., blackbox instantiators (YMS Cocktail instantiator).
E5. The specification of a production strategy may be simplified if the sequence of instantiators can be described in terms of a batch job or an instantiation workflow. This supports the definition of the sequence of instantiators in combination with basic operations such as deleting or copying individual files or folders.

These insights provided a motivation for us to derive a domain-specific language for that purpose. We will outline the basics of that language in the next section.

4 Variability Implementation Language

In this Section, we describe the concepts of the INDENICA Variability Implementation Language (VIL). VIL is designed to solve the issues we experienced from the instantiator-based realization discussed in Section 3. Further, VIL realizes the requirements for variability implementation discussed in Section 2 in terms of a generic, unified variability instantiation language.

Actually, VIL is not a single language. It consists of four main constituents:

- **Asset model:** The asset model defines the individual capabilities of various types of assets used in variability instantiation, such as Java source code, Java byte code, XML files but also components (for runtime variabilities), or elements of the file system such as files or folders. Production strategies are operations on the types of the input and output artefacts using the capabilities of the assets for specifying the instantiation.
- **VIL template language** is used to instantiate a certain type of target artefact in a reusable way. Basically, the VIL template language covers generation as well as transformation-based production strategies.
- **Blackbox instantiators:** In some situations it might be difficult, inconvenient or even impossible to describe a production strategy using the VIL template language. One example is the Cocktail instantiator discussed in Section 3.2.1 as it mainly modifies Java bytecode and, thus, it is easier to realize it in an usual programming language such as Java, i.e., from the point of view of VIL as a black box. In case of legacy product lines, an existing instantiator may be called or wrapped into a VIL extension.
- **VIL workflow language:** A language to define individual production strategies, i.e., to relate artefacts and instantiation mechanisms, to combine production strategies to workflows and to specify the execution of workflows in terms of binding times (more generally conditions on IVML attributes).

VIL and its sublanguages rely on existing, practically proven concepts such as workflow or template languages in order to avoid reinventing the wheel. However, existing concepts as well as related tooling does not provide the full support for variability instantiation. Thus, we reuse and extend existing work to apply it to variability realization.

We will discuss the four constituents listed above in the following subsections, namely the artefact model in Section 4.1, the template language in Section 4.2 and the (use of) blackbox instantiators along with the workflow language in Section 4.3. Finally, we will link VIL to the INDENICA case studies in Section 4.4.

4.1 VIL Artefact Model

VIL relies on an explicit artefact model. The purpose of the artefact model is to describe the types of artefacts used in the instantiation, their specific capabilities and to obtain more precise specifications of the variability instantiation. This will

enable warnings and errors already when specifying the instantiation, i.e., while defining VIL workflows and templates.

The VIL infrastructure provides a predefined set of artefact types as well as their individual capabilities. However, for instantiating a specific base platform, more specific artefacts are needed, such as SQL scripts or Spring configuration files as in the WMS case discussed in Section 3.2.2. VIL supports this in two ways, namely via programmatic extensions to introduce additional artefact types and via the specification of capabilities in VIL workflows.

Figure 40 depicts the basic (predefined) VIL artefacts. Basically, we distinguish between composite artefacts, like a Java file, and fragment artefacts, e.g., parts of Java code. Regarding composite artefacts, we further distinguish between file system

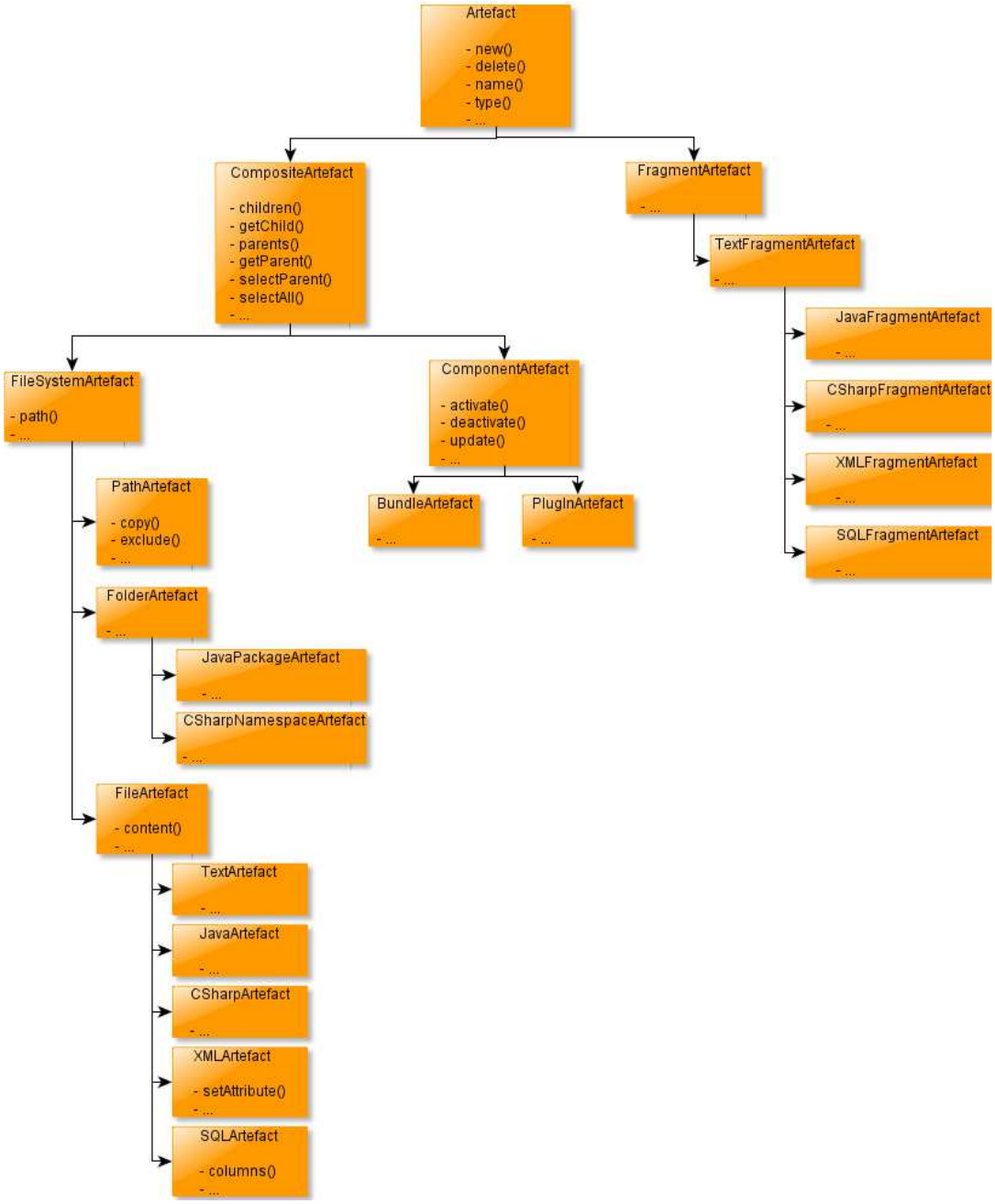

Figure 40: VIL artefact model including default artefacts

artefacts such as paths, files and folders. While file artefacts can be any type of file, folder artefacts subsume other artefact types, like Java packages and C# namespaces as they are typically represented as folders in a filesystem. Software components (component artefacts) can be used for runtime manipulation. Please note that neither all artefact types, nor all operations are shown in Figure 43. We focus here on those artefacts and operations required for describing the instantiation in the INDENICA use cases. Based on this artefact model, VIL will provide a unified way of specifying the instantiation of individual artefacts as we will describe in the following subsections.

4.2 VIL Template Language

The VIL template language aims at a generic description of the instantiation of artefacts. VIL supports:

- **Value insertion**, i.e., at specified locations in the artefact, configured values from a variability model are taken over. The template language must support **value mapping**, as discussed in Section 3.2.3, as in several cases values from the configuration must be mapped to artefact specific values.
- **Alternatives**, i.e., to decide among various alternative instantiations based on values or properties defined in the configuration as done in Section 3.2.2 and 3.2.3. Alternative expressions include logical, relational, mathematical, string and regular expression operations as well as the (extensible) operations defined in the artefact model and in VIL.
- **Iteration** over configured IVML containers (see Section 3.2.2), the structure of IVML compounds (see Section 3.2.2) and locally defined value ranges. In particular, specific iterator operations similar to those for collections in OCL [18, p. 168ff] enable the selection of variables from the variability model (see Section 3.2).
- **Undefined expressions:** Similar to OCL [18], undefined expressions in VIL are prevent further evaluation and, thus, the execution of the containing statement. This simplifies the specification of VIL templates.
- **Named subtemplates** enable the specification reusable transformation tasks and facilitate **template imports**.
- **Mixing** of elements from the target artefacts with statements from the VIL template language to specify the instantiation. Here, statements from the VIL template language are included into mark-ups. VIL supports configurable mark-ups in order to increase the application range, i.e., not to exclude artefact types due to a clash with existing semantics of the VIL mark-ups.
- **Extension capabilities**, i.e., mechanisms that enable openness in VIL, i.e., extension of existing (imported) templates. Therefore, the VIL template language supports polymorphic dispatch similar to Xtend [4].

The VIL template language combines capabilities of popular generator or template languages such as Xtend [4], Xpand [3] and Apache Velocity [27]. Although the VIL template language is rather close to Xtend, it avoids a tight integration with Java concepts in the template language. Further, it integrates (the access to) IVML models with the VIL artefact model, provides instantiation specific operations and enables

the customization of the language in mark-ups (relying on Xtext [5] language infrastructure generation).

Below, we illustrate the VIL template language in the context of the INDENICA Yard Management use case. Figure 41 depicts the derivation of the Cocktail configuration from a given IVML configuration (including the underlying variability model) in terms of a generator template. In Figure 41, lines 1-13 define the main template which is called from the workflow. The input to that section is the IVML configuration (`config`) and the target output artefact (`output`). Within the main template, in line 2 the artefact is created, i.e., the existence of an empty artefact is ensured. Then, the production of the content of the artefact is described (lines 3-12), which mainly consists of Cocktail specific XML elements. As individual variables make up most of the content of the target artefact, a statement for processing the variables is inserted between the artefact prologue (lines 3-6) and the ending of the artefact (lines 9-12). Here, prologue and ending are stated as artefact sections enclosed in `'''`, i.e., target artefact elements which will be part of the `output`. Such sections may also include VIL statements with mark-up as we will show below. In lines 7-9, each variable in `config` is considered and processed by the sub-template `variabilities` (defined in lines 14-18).

```
1  def main(Configuration config, TextArtefact output) {
2    new TextArtefact output
3    '''
4    <?xml version="1.0" encoding="utf-8">
5    <cocktail>
6    '''
7    for $1 in config do
8      variabilities($1)
9    '''
10   </cocktail>
11   '''
12 }
13
14 def variabilities(DecisionVariable variable) {
15   '''
16   <decision name = "#variable.declaration.name#"
17     value = "#variable.value#"/>
18   '''
19 }
```

Figure 41: VIL generation template for Cocktail configuration files

The main purpose of the (reusable) sub-template named `variabilities` is to create a target XML element for each decision and to take over the relevant information from the IVML variability configuration (here we use # as mark-up). The

enclosed expressions are evaluated in the context of the given IVML configuration, i.e., the respective values from the variable declaration and the value of the actual decision variable are taken over into the target artefact.

In addition, the VIL template language also enables the transformation of existing artefacts. Therefore, the main template receives an input artefact as parameter and all operations are then applied to that artefact. Relying on the artefact model, specific elements can be identified, selected and replaced. We will discuss the use of the VIL template language for the transformation in context of the WMS case study in Section 4.4.

4.3 VIL Workflow Language

The VIL workflow language enables the specification of individual production strategies, their combinations as well as their application at various binding times. The VIL workflow language is inspired by existing build process languages such as make [9], ANT [25], Gradle [2] or MWE2 [5, p. 177ff], but it provides specific concepts for binding times, integration of instantiators, and integration with the VIL template language.

The VIL workflow language provides the following concepts:

- **Production strategy:** A production strategy is specified as the application of an instantiator at a specific binding time to a given set of input artefacts in order to produce a certain set of output artefacts. Instantiators may be given as VIL-integrated instantiators such as the processor for the VIL template language and blackbox instantiators. The main difference is that VIL-integrated instantiators (including wrapped blackbox instantiators) provide information on the input and output artefact types to the VIL workflow engine while pure blackbox are just executed.
- **Alternatives:** Selection of production strategies based on conditions in terms of variables and values defined in the IVML configuration (including the underlying variability model).
- **Loops:** Repeated execution of production strategies based on a certain container in the IVML variability configuration, the structure of IVML elements such as compounds or selected subsets of the configuration, respectively.
- **Workflow:** A workflow is a named, reusable sequence of production strategies which operates on a given set of input and output artefacts. A workflow may specify prerequisites in terms of required binding times, artefacts / artefact types or conditions on subsets of the variability model. Due to the known input-output relation for the individual productions strategies (as exposed by the instantiators), the VIL workflow engine can validate the application of production strategies within a workflow and emit warnings and errors already while the VIL workflow is specified. To facilitate reuse, a VIL workflow may import other VIL workflows.
- **Workflow runner:** The definition of binding times in IVML is rather generic. Binding times are defined as attributes [10], i.e., neither the name of the attribute nor the (user-specified) type are fixed. Thus, VIL needs a

specification of the semantics of binding times, i.e., the name of the attribute(s) and the intended sequence of binding times. This is done in the workflow runner (section).

- **Linking the configuration:** A VIL workflow specifies the actual IVML model it relies on for passing it to VIL templates, using it in conditions and for specifying the sequence of binding times in the workflow runner section.
- **Integration into the build process:** VIL may be executed from EASy-Producer or from a build process (cf. requirement I11 in D2.1 [10]). The latter is supported through specific integrations of parsing and executing VIL specifications into existing build process languages. However, VIL workflows are not designed to specify the entire build process for a service-based system, a base platform or, in general, a software product line as this would require a unification of various build process languages such as make [9], Maven [26], Gradle [2], msbuild [16], ANT [25], MWE2 [5, p. 177ff], etc.

Below, we illustrate the VIL workflow language in the context of the INDENICA Yard Management use case also including the application of the example VIL template discussed in Section 4.2.

Figure 42 depicts a fragment from the VIL workflow for the YMS use case. In line 1 it imports version 0.5 of the YMS configuration (including variability model) into `config`. This establishes the link to the configuration similar to model imports in IVML [10], i.e., it also supports the evolution of variability models and configurations.

```
1  import YMSCfg with YMS.version=v0.5 into config
2
3  runner (PathArtifact source, PathArtifact target,
4    ParameterSet param) {
5      apply(bindingTime=compile)
6      using (source, target) on param
7  }
8
9  select bindingTime=compile for
10 workflow compileFlow (PathArtifact source,
11   PathArtifact target) {
12   copy(source, target).exclude(”$source/templates”)
13   tmp = vilTemplateProcessor(
14     config, ”$target/cocktail.xml”,
15     ”$source/templates/cocktail.vtl”)
16   cocktailInstantiator(tmp)
17 }
18 // runtime workflow skipped
```

Figure 42: Fragment from the YMS production strategy

Lines 3-7 specify the workflow runner. Basically, the runner receives a source and a target path. The source path contains the generic artefacts, while the target path denotes where the instantiated artefacts shall be placed. The target path may be empty as basic file operations such as the creation of paths or files are handled transparently by the VIL workflow engine. The `apply` command in lines 5-6 binds names of attributes to attribute values, here, the attribute `bindingTime` to the values `compile`. Corresponding workflows will be executed using the parameters `source` and `target` provided to the runner. In addition, `param` enables to pass arbitrary name-value pairs to the VIL workflow so that the caller can request the instantiation for a certain binding time. For example, if the caller passes the name-value pair `bindingTime=compile`, then only the compile time workflows will be executed.

The YMS workflow for compile time is depicted in Figure 42 in lines 9-17. The workflow named `compileFlow` is executed only if the runner enables `compile` time as `bindingTime`. In addition to this binding-time-specific selection condition, further conditions on the configuration and the artefacts may be specified, such as whether certain variables are defined. A workflow is executed only if all selection conditions are fulfilled. The first step in the workflow in line 12 copies all source artefacts to the target path except for artefacts in `"$source/templates"`. In VIL, paths may be denoted as strings containing VIL variables indicated by the `$` prefix (similar to ANT and MWE2). Please note that copy operations depend on the type of artefact and may be overridden by VIL extensions or operations defined in the VIL workflow (not shown in Figure 42). This enables specific copy operations such as the package adaptation for Java artefacts discussed in Section 3.1.1. Then, in line 13, the VIL template processor is executed applying the IVML configuration and the specified target file artefact to the given template. The resulting artefact is stored in the VIL variable `tmp`. Finally, in line 16 the blackbox Cocktail instantiator (cf. Section 3.2.2) is executed on the previously generated configuration in `tmp`. Here, the Cocktail instantiator is made available to VIL through wrapper code, i.e., the execution of the Cocktail instantiator looks akin to the template processor as well as the specific input-output relation implemented by Cocktail (XML configuration to transform Java classes) is provided to the VIL workflow engine.

For executing the workflow given in Figure 42, either EASy-Producer starts the VIL workflow engine on user request or this is done from a build process. An example for the integration into an ANT build process is illustrated in Figure 43. Here, the VIL workflow specification (`wms.vil`), the source platform project and the target platform project are passed as input to the VIL execution engine. In addition, a parameter is given, which requests instantiation for compile time.

```
<!--ANT prologue and other targets skipped-->

<target name=”compile”>
  <processVIL
    file=”YMSplatform/templates/wms.vil”
    source=”YMSplatform”
    target=”YMSinstance”
      <param name=”bindingTime” value=”compile”/>
  </processVIL>
  <!--remaining tasks such as javac skipped -->

</target>

<!--remaining ANT elements skipped -->
```

Figure 43: Fragment from an ANT build file integrating the VIL workflow

4.4 Application to INDENICA Case Studies

In this section, we will discuss the application of VIL to the INDENICA case studies.

4.4.1 Yard Management System

The application to the YMS case study was already discussed as running example in Section 4.2 and 4.3.

4.4.2 Warehouse Management System

In this section, we will discuss the application of VIL to the WMS case study. As described in Section 3.2.2, the instantiation for the WMS subsystem is realized as four different instantiators. Please note that we will not repeat the ideas and algorithms of these instantiators here as we discussed them in Section 3.2.2. In this section, we will focus on the specification of the WMS instantiation in terms of VIL. We will first discuss the templates that instantiate the individual artefacts of the WMS. With these templates in place, we will then describe the complete workflow.

The Spring configuration file instantiation takes an existing Spring configuration file as input and modifies marked elements based on a given (IVML) configuration. Figure 44 shows this production strategy in terms of a VIL template. In the main template an iteration over all XML nodes with child “`easy:varprop`” in the given XML artefact `target` (line 2) is defined. If an IVML variable corresponding to the contents of the child “`easy:varprop`” exists (line 3-5), the value of the attribute `value` in the parent node is replaced by the actual value of the configuration variable (line 6) and the child node is removed from the target artefact (line 7). Lines 12-14 and the corresponding call in line 6 are actually not needed in the WMS case study but enable reuse in the RMS case study in Section 4.4.3. Therefore, we also skipped the WMS specific focus on the `properties` element without affecting the correctness of the resulting artefact.

```
1  def main(Configuration config, XMLArtefact target) {
2    for $1 in target.selectParent(”*/easy:varprop”) do {
3      var child = $1.getChild(”easy:varprop”)
4      var var = config.variables.get($child.content)
5      if exists(var) {
6        $1.setAttribute(”value”, map(var.value))
7        child.delete()
8      }
9    }
10 }
11
12 def map(Value value) {
13   value
14 }
```

Figure 44: VIL generation template for Spring configuration files.

The database schema instantiation in Figure 45 modifies the database schema of the WMS according to the (IVML) configuration. In line 2 of the template, a selection of the relevant variables according to the WMS naming convention (the VIL execution engine provides only access to frozen variables) is defined. Then, for each “CREATE TABLE” statement in the SQL target artefact (line 3) the relevant variables are narrowed down to those variables for which the compound name matches the table name (line 4). Finally, all columns of the containing SQL statement (line 5) are considered and for those having a corresponding IVML variable (in the pre-digested set `cfg1`) with value `false` (line 6-7), the respective column is deleted (line 8).

```
1  def main(Configuration config, SQLArtefact target) {
2    var cfg = config.select(v|v.name.endsWith(”Config”))
3    for $1 in target.selectAll(”CREATE TABLE”) do {
4      var cfg1 = cfg.select(v|v.compoundName = $1.name)
5      for $2 in $1.columns() do
6        for $3 in cfg1.select(v|.value = false &&
7          v.name + ”Config” = $2.name) {
8          $2.delete
9        }
10     }
11   }
12 }
```

Figure 45: VIL generation template for database schemas.

The object-relational (OR) mapper configuration instantiation in Figure 46 removes those mappings, which have been disabled in the (IVML) variability configuration. In the main template (lines 1-7), a selection of the relevant variables according to the WMS naming convention is defined. Then, a sub template (lines 9-21) for the individual sections of the OR mapper configuration is applied (only two specific applications are shown in Figure 46). In the sub-template, a selection of the `parent` element according to given element path (line 11) is defined and the child elements are processed (line 14-19) only if a corresponding IVML variable exists (line 12-13). If the IVML variable related to the child node exists and has value `false` (line 15-16), the child node is deleted (line 17).

```
1  def main(Configuration config, XMLArtefact target) {
2    var cfg = config.select(v|v.name.endsWith(”Config”))
3    process(cfg, target, ”Schema/EntityType”, ”Property”)
4    process(cfg, target, ”Mapping/EntityContainerMapping”
5      + ”/MappingFragmet”, ”ScalarProperty”)
6    // further processing according to structure
7  }
8
9  def process(Configuration config, XMLArtefact target,
10   String parent, String child) {
11   for $1 in target.selectAll(parent) do {
12     var var = config.get (
13       $1.getAttribute(”name”) + ”Config”)
14     for $2 in $1.selectAll(child) do {
15       var var2 = var.getVariable($2.name)
16       if (var2.value = false)
17         $2.delete()
18       }
19     }
20   }
21 }
```

Figure 46: VIL generation template for OR-mapper configuration files.

The last VIL template for instantiating the WMS describes the creation of a SQL database initialization script for initializing the database with the actual content. In the template in Figure 47 a complete initialization script is created from scratch. In the core template `main` (lines 1-12), the creation of the basic script parts, like the beginning and the end of the script (indicated by comments) is defined. The creation of the insert statements is defined in the sub-template `instantiate` (lines 14-34), which is applied for each nested element of a compound variable whose name ends with “`Topology`” (line 6, akin to the instantiator mechanism described in Section 3.2.2). In the sub-template, first the creation of the basic insert statement with the name of the compound variable (without postfix “`Topology`” to match the names of

the database tables) is defined. The nested elements of this compound variable represent the actual content. Each variable corresponds to a column or its value, respectively. Following the SQL syntax for insert statements, initialization is created in two loops, one for the column names (lines 20-23) and one for the related values (lines 27-30).

```
1  def main(Configuration config, SQLArtefact output) {
2    new SQLArtefact output
3    ’’’
4    // Beginning of the script
5    ’’’
6    for $1 in config.select(v|v.name.endsWith(”Topology”)
7      && v.typeOf(”Compound”)) do
8        instantiate(output, $2)
9    ’’’
10   // End of the script
11   ’’’
12 }
13
14 def instantiate(SQLArtefact output,
15     DecisionVariable variable) {
16   ’’’
17   INSERT [dbo].[#variable.declaration.name.substring(
18     -”Topology”.length())#] (
19   ’’’
20   for $1 in variable.nestedElements do
21     ’’’
22     [#$1.declaration.name#] ,
23     ’’’
24   ’’’
25   ) VALUES (
26   ’’’
27   for $1 in variable.nestedElements do
28     ’’’
29     N’#$1.declaration.value#’ ,
30     ’’’
31   ’’’
32   )
33   ’’’
34 }
```

Figure 47: VIL generation template for database content initialization scripts

Finally, Figure 48 shows the VIL workflow for the WMS use case. The runner section is empty as the WMS variability does not separate between compile and binding time. Thus, the workflow is specified without selection condition. The workflow copies the artefacts (akin to YMS workflow described in Section 4.3) and executes then the four VIL templates discussed above.

```
1  import WMSCfg with WMS.version=v0.5 into config
2
3  runner (PathArtifact source, PathArtifact target,
4    ParameterSet param) {
5    apply using (source, target)
6  }
7
8  workflow compileFlow (PathArtifact source,
9    PathArtifact target) {
10   copy(source, target).exclude(”$source/templates”)
11   vilTemplateProcessor(
12     config, ”$target/**/*.config”,
13     ”$source/templates/easy_varprop.vtl”)
14   vilTemplateProcessor(
15     config, ”$target/db/schema.sql”,
16     ”$source/templates/sql_schema.vtl”)
17   vilTemplateProcessor(
18     config, ”$target/db/db.edmx”,
19     ”$source/templates/or_mapping.vtl”)
20   vilTemplateProcessor(
21     config, ”$target/db/data_init.sql”,
22     ”$source/templates/data_content.vtl”)
23 }
```

Figure 48: Fragment from the WMS production strategy

4.4.3 Remote Maintenance System

In this section, we will show the application of VIL in the RMS case study based on the instantiators discussed in Section 3.2.3.

The RMS service configuration instantiation maps the decision variables and their values given in terms of an IVML configuration to the properties of an XML file (the RMS service configuration). The main difference between the Spring production strategy shown in Figure 44 in Section 4.4.2 and the RMS strategy is that in the RMS case the configuration values must be mapped explicitly. Thus, the RMS production strategy can extend the (generic) Spring production strategy in Figure 44 by providing a specific value mapping.

```
1 import easy_varprop.vil
2
3 def map(Value<ServiceState> value) {
4   switch (value) {”full” = ”multipoint”,
5     ”point2point” = ”p2p”,
6     value}
7 }
```

Figure 49: VIL generation template for RMS configuration files.

In line 1 of Figure 49, the base template is imported which includes the `main` and the `map` sub template. Then the map sub template is overloaded in lines 3-7 using the specific type and a shorthand switch-case statement.

Figure 50 illustrates the VIL workflow for the RMS use case. In line 1 it imports the RMS variability configuration. In lines 3-7 the workflow runner is configured for compile and runtime binding in the given sequence. The compile time workflow (lines 9-17) copies the artefact and executes the RMS configuration file template shown in Figure 44. Akin to the instantiator shown in Section 3.2.3, the runtime production workflow (lines 19-16) exports the constraints of the runtime variabilities in terms of Drools rules. Therefore, it relies on a wrapped blackbox instantiator, which integrates the EASy-producer Drools reasoning component into VIL. In line 22, the imported configuration is instantiated for all variabilities prior to runtime. The instantiation result is exported as Drools rules in line 23. The target artefact is finally instantiated by writing the exported rules into a text file artefact. A slightly modified workflow can store the exported rules into the INDENICA infrastructure repository (not shown in Figure 44).

```
1  import RMSCfg with RMS.version=v0.3 into config
2
3  runner (PathArtifact source, PathArtifact target,
4    ParameterSet param) {
5    apply(bindingTime=compile, bindingTime=runtime)
6      using (source, target) on param
7  }
8
9  select bindingTime=compile for
10 workflow compileFlow (PathArtifact source,
11   PathArtifact target {
12   copy(source, target).exclude(”$source/templates”)
13   vilTemplateProcessor(
14     config, ”$target/rms.cfg”,
15     ”$source/templates/rms.vtl”)
16   // compile etc.
17 }
18
19 select bindingTime=runtime for
20 workflow runtimeFlow (PathArtifact source,
21   PathArtifact target) {
22   tmp = DroolsReasoner (config, ”-instantiate”, runtime)
23   rules = DroolesReasoner(tmp, ”-getOpenRules”)
24   new TextArtifact target
25   target.write(rules)
26 }
```

Figure 50: Fragment from the VIL workflow for the RMS case study

4.5 Summary

The Variability Implementation Language (VIL) introduced in this section is a major step forward in defining and managing the application of variability implementation techniques. Based on the previous results in realizing production strategies and the experience gained in INDENICA and other projects (cf. Section 3.3, E1-E5), VIL improves the current realization of the INDENICA variability concepts in terms of a generic, unified variability instantiation language. In this regard VIL is unique at this point:

- VIL supports the instantiation of various variable artefacts ranging from the creation of new files from scratch to the modification of individual artefact fragments in place (E1). The asset model of VIL provides distinct capabilities to manipulate artefacts of different types and granularity. Further, this model

ensures type-compliance when applying a certain operation to a specific artefact, guaranteeing that the operation is valid for the type of artefact. The open plug-in infrastructure of the artefact model also enables the definition of new artefact types and the specification of valid operations for the types.

- The VIL template language provides capabilities to define (reusable) translations of (parts of) the variability model into the result artefacts in a generic way (E2). While this language provides its own language elements and operations to define basic translation-mechanisms (conditions, iterations, etc.), it further reuses operations of the IVML, e.g., accessing the value of a decision variable, and the asset model, e.g., setting the value of an attribute in a XML-file. These translations also support the realization of runtime variability (E3), e.g. in terms of translating values of the variability model into an artefact type-specific language.
- The VIL workflow language enables the integration and execution of external tools and mechanism, like the black-box instantiators described in Section 3, as part of an instantiation (E4). While instantiators are typically registered to EASy-Producer (cf. Section 3.1.1) and, thus, can be called simply by their names (as shown in Figure 42 for the Cocktail Instantiator), VIL provides language elements for calling other (unknown) external tools. Further, VIL supports the configuration of such tools in terms of passing parameters, similar to calling a tool via the Windows command console.
- The VIL workflow language also enables the definition of sequences in which instantiators, external tools, and basic operations like file system operations must be executed (E5). The order of execution is defined by the order of the operations within a workflow in VIL.

VIL supports all realized mechanism for variability instantiation in INDENICA and provides further extensions to the definition of such instantiations. The implementation of this language and the corresponding constituents will replace the Instantiation Core of the variability engineering tool EASy-Producer (cf. Section 3.1.1) in the future to provide a flexible and easy-to-use mechanism for specifying variability implementation in INDENICA.

5 Conclusion

In this deliverable, we discussed the current approach to variability implementation in INDENICA, its application in the INDENICA case studies and its evolution based on lessons learned so far.

In a first step, we summarized the requirements based on the already existing requirements collection in D2.2.1. In this deliverable, the focus is on highlighting the aspects to be realized by a variability implementation approach.

In Section 3, we discussed the current state of the realization in two parts, the technical framework provided by EASy-Producer, the INDENICA variability management tool, and its actual application to the INDENICA case studies. Actually, EASy-Producer can be extended by so called instantiators, which perform specific instantiation tasks. Currently, we exploit this approach to realize the variability implementation in the individual case studies. In summary, this implementation-based approach fulfils the requirements for variability instantiation in INDENICA, but it also lacks flexibility.

In Section 4, we introduced the Variability Implementation Language VIL, a generic and unified approach to variability implementation, which extends the current approach. VIL consists of an explicit artefact model, a template language and a workflow language, both based on related languages such as Xtend or MWE2, respectively. Based on a discussion of the concepts of VIL, we illustrated the application of VIL in terms of variability instantiation specifications for the INDENICA case studies.

As a result, the language concepts of VIL enable a flexible and easy specification of variability implementations. The realization of VIL as a replacement for the instantiation core in EASy-Producer is on the way. It will be realized as part of Deliverable D2.4.2. However, the current language statements as well as the artefact-specific operations were selected from the fundamental requirements of the INDENICA use cases. This selection will be subject to future refinements in order to further simplify the variability instantiation, e.g., by more specific artefact operations. This refinement will take place after applying VIL also in related projects and reviewing the resulting specifications.

References

[1] BigLever Software, Inc. BigLever's Gears Product Line Engineering Tool and Lifecycle Framework, 2013. Online available at: http://www.biglever.com/solution/product.html.

[2] Hans Dockter Adam Murdoch. Gradle User Guide Version 1.4, 2012. Online available at: http://www.gradle.org/docs/current/userguide/userguide.pdf.

[3] Eclipse Foundation. Xpand, 2013. Online available at: http://projects.eclipse.org/projects/modeling.m2t.xpand.

[4] Eclipse Foundation. Xtend 2.4.0 User Guide, 2013. Online available at: http://www.eclipse.org/xtend/documentation/2.4.0/Documentation.pdf.

[5] Eclipse Foundation. Xtext 2.4 Documentation, 2013. Online available at: http://www.eclipse.org/Xtext/documentation/2.4.0/Documentation.pdf.

[6] H. Eichelberger, C. Kröher, K. Schmid. Variability in Service-Oriented Systems: An Analysis of Existing Approaches. *Proceedings of the 10th international conference on Service-Oriented Computing (ICSOC'12)*, 516–524, 2012.

[7] S. El-Sharkawy, C. Kröher, K. Schmid. Supporting heterogeneous compositional multi software product lines. I. Schaefer, I. John, K. Schmid, editors, *Proceedings of the Joint Workshop of the 3rd International Workshop on Model-driven Approaches in Software Product Line Engineering and the 3rd Workshop on Scalable Modeling Techniques for Software Product Lines (MAPLE/SCALE 2011) at the 15th International Software Product Line Conference (SPLC '11)*, volume 2. ACM, 2011.

[8] Y. Eldogan. *Automatisiertes Deployment einer SCA-basierten Anwendung in die Cloud*. PhD thesis, University of Hildesheim, Institute for Computer Science, 2012. (in German).

[9] Free Software Foundation. GNU Make - A Program for Directing Recompilation - Version 3.82, 2010. Online available at: http://www.gnu.org/software/make/manual/make.pdf.

[10] INDENICA Consortium. Deliverable D2.1: Open Variability Modelling Approach for Service Ecosystems. Technical report, 2011.

[11] INDENICA Consortium. Deliverable D2.2.1: Variability Implementation Techniques for Platforms and Services (interim). Technical report, 2011.

[12] INDENICA Consortium. Deliverable D4.1: Framework for Deployment, Monitoring & Controlling of Virtual Service Platforms. Technical report, 2011.

[13] INDENICA Consortium. Deliverable D5.1: Description of Feasible Case-Studies. Technical report, 2011.

[14] INDENICA Consortium. Architecture for Role-Based Governance of Virtual Service Platforms, 2012. Online available at: http://indenica.eu/fileadmin/INDENICA/user_upload/d32-archgov.pdf.

[15] INDENICA Consortium. Deliverable D2.4.1: Variability Engineering Tool (interim). Technical report, 2012.

[16] Microsoft. MSBuild Reference, 2013. Online available at: http://msdn.microsoft.com/en-us/library/0k6kkbsd%28v=vs.90%29.aspx.

[17] Microsoft. Visual Studio Solution (.sln) File, 2013. Online available at: http://msdn.microsoft.com/en-us/library/bb165951%28v=vs.80%29.aspx.

[18] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.3.1 2012-01-01, Object Management Group, 2012. Online available at: http://www.omg.org/spec/OCL/2.3.1/PDF/.

[19] OSGI Alliance. OSGi Release 5 Specification, 2012. Online available at: http://www.osgi.org/Specifications/HomePage.

[20] pure-systems GmbH. pure::variants User's Guide - Version 3.0 for pure::variants 3.0, 2012. Online available at: http://www.pure-systems.com/fileadmin/downloads/pure-variants/doc/pv-user-manual.pdf.

[21] M. Rosenmüller N. Siegmund. Automating the configuration of multi software product lines. *Proc. of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VAMOS'10)*, 123–130, 2010.

[22] K. Schmid H. Eichelberger. EASy-Producer – A Product Line Production Environment. *Proceedings of the 12th International Software Product Line Conference (SPLC '08)*, 357–357, 2008.

[23] K. Schmid H. Eichelberger. Model-Based Implementation of Meta-Variability Constructs: A Case Study using Aspects. *Proceedings of the 2nd International Workshop on Variability Modelling of Software-intensive Systems (VAMOS '08)*, 63–71, 2008.

[24] SpringSource. Spring Projects - Spring Framework, 2013. http://www.springsource.org/spring-framework.

[25] The Apache Software Foundation. Apache Ant 1.8.2 Manual, 2013. Online available at: http://ant.apache.org/manual/index.html.

[26] The Apache Software Foundation. Apache Maven Project, 2013. Online available at: http://maven.apache.org/index.html.

[27] The Apache Software Foundation. The Apache Velocity Project, 2013. Online available at: http://velocity.apache.org/.