Engineering Virtual Domain-Specific Service Platforms

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

Variability Engineering Tool (interim)

Abstract

Domain-specific customization of service platforms can only be effective with appropriate tool support. In this deliverable we will present the current state of the INDENICA Variability Engineering tool, by illustrating its application as well as its technical realization. Regarding the application, we will discuss an introductory example showing the individual steps of customizing a service platform. Regarding the technical realization we will discuss the architecture and its individual components along with their realization status and, finally, the interaction and integration with the work done in the other INDENICA work packages.

Document ID:	INDENICA – D2.4.1
Deliverable Number:	D2.4.1
Work Package:	WP2
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2012-10-02
Author(s):	SAP, SUH

Project Start Date: October 1st 2010, Duration: 36 months

Version History

0.1	29. Jun 2012	initial version
0.2	03. Sep. 2012	running example added
0.3	18. Sep 2012	variability engineering tool design and initial relation to other WPs added
0.4	01. Oct. 2012	final revision and corrections of complete document
1.0	02. Oct. 2012	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. However, customization of software in general and of service platforms in particular can only be effective with appropriate tool support. In this deliverable we discuss the current state of the tool support designed and realized in INDENICA. The work presented here is based on previous results and concepts researched in work package 2, namely the classification of variability implementation techniques (D2.2.1) and, in particular, the design of the INDENICA Variability Modelling Language (IVML) in Deliverable D2.1.

In Section 2 we will present the variability engineering tool from a practical perspective using an introductory example. This example will build on the service platform example introduced in Deliverable D2.1 and will show how to use the tool to derive a domain-specific service platform variant. For this purpose, we discuss all steps needed to successfully define a service (base) platform as a product line project, including variability modelling and implementation. Starting from these artefacts, we will describe how to derive a domain-specific service platform including its configuration, validation and instantiation.

Section 3 will focus on the technical aspects and the current state of the implementation. First, we will describe the architecture of the variability engineering tool in terms of its logical components and its overall extension capabilities. Then we will discuss the realization status of the major components like the IVML object model and the instantiation support. In particular, in Section 3.3 we will detail the reasoning mechanism for IVML, i.e., the selection of the underlying reasoning technology as well as the results of initial performance experiments.

In Section 4 we will discuss the status of the integration of the variability engineering tool with the work done in the other work packages. This section will give an overview on the integration with the INDENICA development tools (WP1, WP3), with the runtime environment (WP4) as well as with the service platform case studies (WP5).

Finally, in Section 5 we will review the overall status of the realization in WP2 and conclude by describing further realization and optimization work to be done in the third year of the project.

2 Running Example

In this section, we will focus on the roles of a Platform Provider and a Platform Variant Creator (see Deliverable D3.2) in order to illustrate the usage of EASy[1]-Producer based on the running example introduced in Deliverable D2.1. We will prototypically model and implement the variability of a content-sharing platform, which allows the user to upload, annotate, release and share content of various types. In Section 2.1, we will adopt the role of a Platform Provider and describe the definition of a software product line from which multiple variants of the content-sharing platform can be derived. This includes the definition of the variability model using the INDENICA Variability Modelling Language (IVML) and the implementation of these variabilities in source code. In Section 2.2, we will adopt the role of a Platform Variant Creator and describe the derivation of a specific service platform variant including the variant configuration and the instantiation of the corresponding artefacts.

We will use the following font styles throughout this section to illustrate and distinguish between actions, active tool elements, and added input:

- EASy-Producer (as well as Eclipse) provides multiple editors, wizards, etc. In order to identify the **active tool element** currently in use, it will be highlighted using bold font.
- All *actions* that will be performed will be highlighted using italics font.
- All input to EASy-Producer will be illustrated in `Courier New`.

2.1 Defining a New Base Service Platform

In this section, we will describe the process of defining the variability of a (base) service platform (a software product line) using EASy-Producer from the perspective of a Platform Provider. We will start with the creation of a new product line project in EASy-Producer, define the configuration space in terms of an IVML variability model, and implement the variabilities using a variability implementation technique. The resulting base service platform (the product line project) will be the basis for the derivation of different content-sharing platforms by a Platform Variant Creator.

The first step towards a product line definition in EASy-Producer is to define a new product line project. For this purpose, start the Eclipse application with the already installed EASy-Producer tool[2]. Start the **New Project Wizard** by opening *File* → *New* → *Project*. Expand the EASy-Producer category and select the entry **New EASy-Producer Project**. This opens the **Product Line Project Wizard** that requires the definition of a name for the new product line project. In our example, we will use `PL_Content_Sharing` as the name of our prototypical product line. Enter the name and click the *Finish* button. The product line project will be created and EASy-Producer will automatically open the **Product Line Editor** as illustrated in Figure 1.

[1] Engineering Adaptive Systems

[2] Information regarding the installation of EASy-Producer can be found in the EASy-Producer User Guide: http://projects.sse.uni-hildesheim.de/easy/docs/guide.pdf

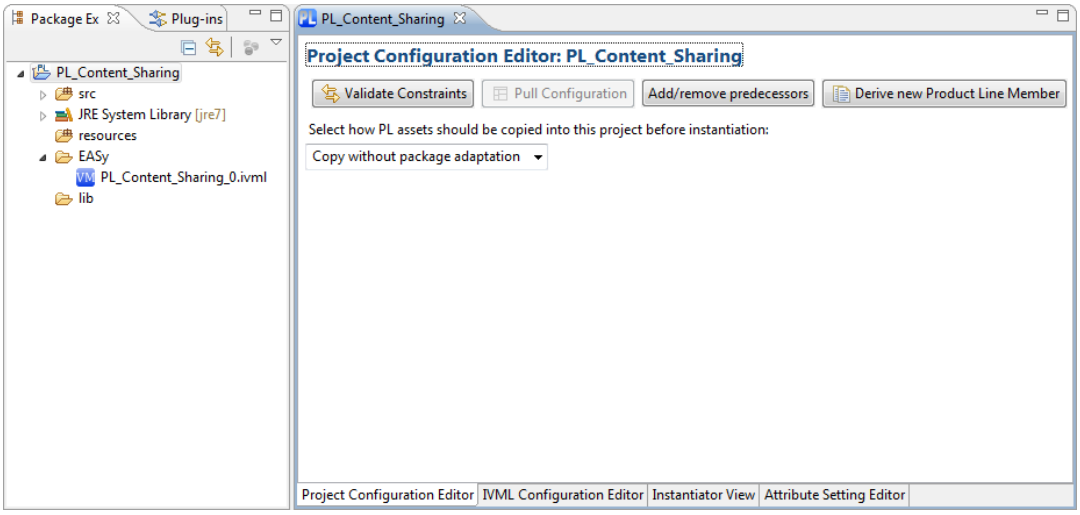

Figure 1: The product line editor in EASy-Producer.

The **Product Line Editor** is the central editor in EASy-Producer as it provides the basic information about a software product line (or a product) as well as the capabilities to derive, configure, and instantiate a product using the different tabs shown in Figure 1. For this purpose, the configuration space (variability model) and the implementation space (variability implementation) must be defined. We will describe both definitions in detail in the next two subsections.

2.1.1 Configuration Space Definition

A variability model defines the valid configuration space of a specific software product line. The variabilities are implemented in the artefacts. In EASy-Producer, we use IVML for defining a variability model and, thus, the configuration space of the content-sharing platform. This model will be the basis for configuring individual service platforms in terms of defining valid value combinations for the configuration space elements (the IVML decision variables).

In EASy-Producer, each product line project comes with its own IVML-file, which can be opened and edited using the **IVML-Editor**. The IVML-file is located in the **EASy-folder** of the project. The name of the file is composed of the name of the product line and the version number (here initially “0”). In our example, double-click the file **PL_Content_Sharing_0** in order to open the **IVML-Editor**.

By default, each IVML-file has a mandatory project element and a mandatory version number as shown in Figure 2. The project element is the top-level element of each IVML file and identifies the configuration space of a certain software project (product line or product). The version element defines the current state of evolution of a project and, thus, identifies a specific (state of a) project. The default version is “v0”.

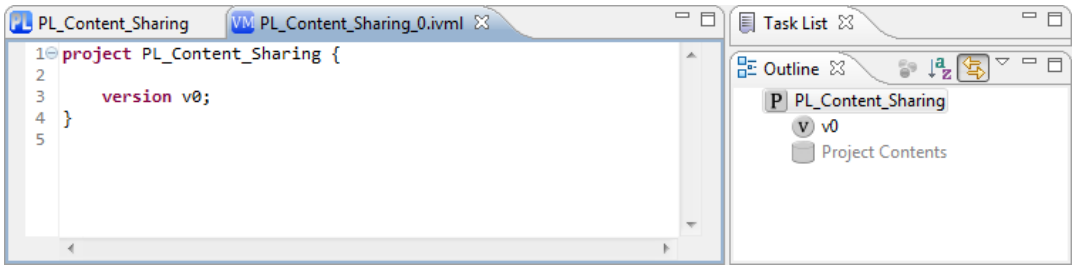

Figure 2: Default IVML file in EASy-Producer.

We characterize the configuration space of the variant-enabled content-sharing base platform by specifying the variability model in IVML. Figure 3 shows a snippet of the variability model (cf. D2.1 for details). First, we define several enumerations that represent the different content types, container types, etc., which an application may support in general (lines 5-8). These enumerations are the basis for specifying the type, for example, of a specific content (lines 10-12). The basic content compound must be refined in order to represent the specific configuration options for `Video`, `3D` (`ThreeD`), and `BLOB` contents (lines 14-27). The other compounds are modelled according to the running example (cf. D2.1 for details). As indicated in the outline on the right side of Figure 3, the two types `Application` and `TargetPlatform` include decision variables of the previously defined (compound) types representing the complete set of configuration options for the content sharing base platform. Thus, two variables (one of type `Application` and one of type `TargetPlatform`) are defined as the main decision variables for configuring a specific content-sharing platform variant. These variables will also be displayed in the **IVML Configuration Editor** tab of the **Product Line Editor**. We will discuss this editor in detail in the process of product configuration in Section 2.2.1.

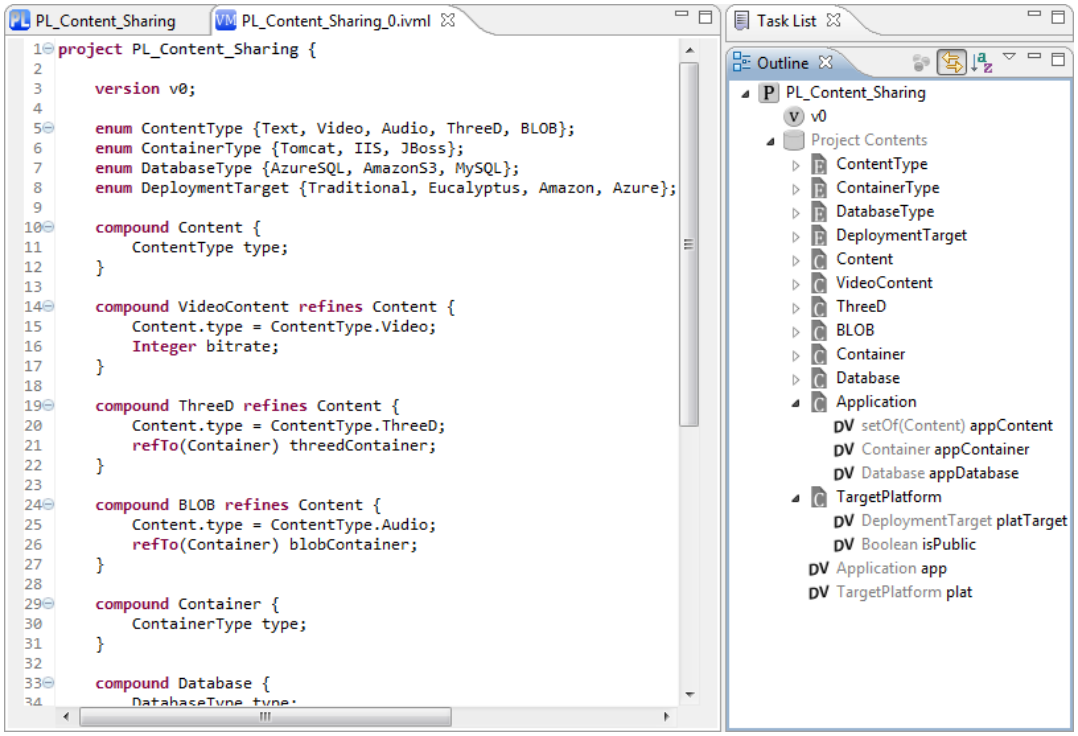

Figure 3: Variability model of the content-sharing application (snippet).

Finally, the variability model, and, thus, the configuration space of the content-sharing application is defined. We will use this model in Section 2.2.1 for configuring a specific content-sharing platform variant. However, in the next section we will first discuss the implementation of the variabilities. This includes the relation of the decision variables to the implementation in order to automatically instantiate different platform variants.

2.1.2 Implementation Space Definition

The implementation space of a specific software product line represents all variable artefacts that can be instantiated according to a specific configuration. The actual implementation of these artefacts depends on the applied variability implementation techniques. A variability implementation technique is a specific approach to realize variability, e.g., using pre-processor directives, aspects, or any of the other techniques described in Deliverable D2.2.1. In EASy-Producer different variability implementation techniques can be applied and combined. However, each variability implementation technique is realized by an individual instantiator, which actually applies the variability implementation technique (we will discuss the instantiators in detail in Section 3.4).

In the running example, we will use SAP's Cocktail instantiator, which was specifically developed for the SAP yard management use case. Cocktail is a variability implementation technique, which supports specific requirements for agile software development of cloud-based service platforms, e.g., to be executed in the SAP Netweaver Cloud[3]. Basically, Cocktail can be applied to any programming language, which supports meta-information in source code. In this example we will demonstrate Cocktail for base platforms developed in Java and express meta-information in terms of Java source code annotations. These annotations are used to bind configuration space elements (decision variables given in terms of their qualified names) to so-called variation points in the source code. A variation point is a "hook" for variability decisions to be bound during the configuration process (cf. Section 2.2.1). Figure 4 illustrates these annotations at attributes in the `Platform` class of the running example, i.e., the actual value of the annotated attribute will change according to the configuration at compile or runtime.

[3] For detailed information regarding SAP Netweaver Cloud visit: http://scn.sap.com/community/developer-center/cloud-platform

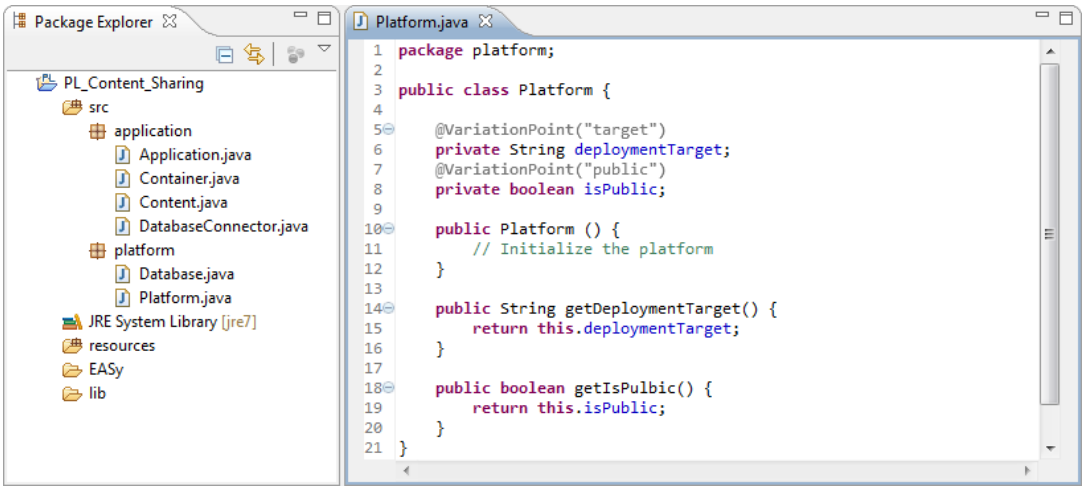

Figure 4: Example variability implementation using SAP's Cocktail approach.

The next step is to define the instantiator for instantiating the artefacts. Open the **Product Line Editor** by *right clicking on the product line project* and select *Edit Productline* in the context menu. Switch to the **Instantiator View** tab of the **Product Line Editor,** which is illustrated in Figure 5. In our example, we will select **CocktailTransformer** as the instantiator for the entire product line by *selecting the corresponding entry* in the drop-down menu in the upper part of the **Instantiator View**. Clicking the *Add Instantiator* button will add the Cocktail instantiator to the product line project. Select the new (instantiator) entry in the list on the left side of the view. This will display the currently selected artefacts that the Cocktail instantiator will instantiate. Of course, at this point we have to select the artefacts by checking the *checkboxes* for all files in the **src** folder. Please note that it is also possible to define multiple instantiators, where each one may be in charge of a subset of the product line artefacts, or define multiple instantiators for the same files. The latter case is required if a single artefact is implemented using multiple, different variability implementation techniques, i.e. to enable some variable parts to be resolved at compile time first (for example, using pre-processer directives), while other part can be resolved later at runtime (for example, using aspects). In such a case, the *Calculate files that will be instantiated multiple times* button can be used to identify files that will be instantiated by more than one instantiator.

Please note that this way of describing the instantiation process will be refined by the Variability Implementation Language (VIL), which is currently under development and its concepts will be discussed in Deliverable D2.2.2. We already introduced the basic ideas of VIL in Deliverable D2.2.1.

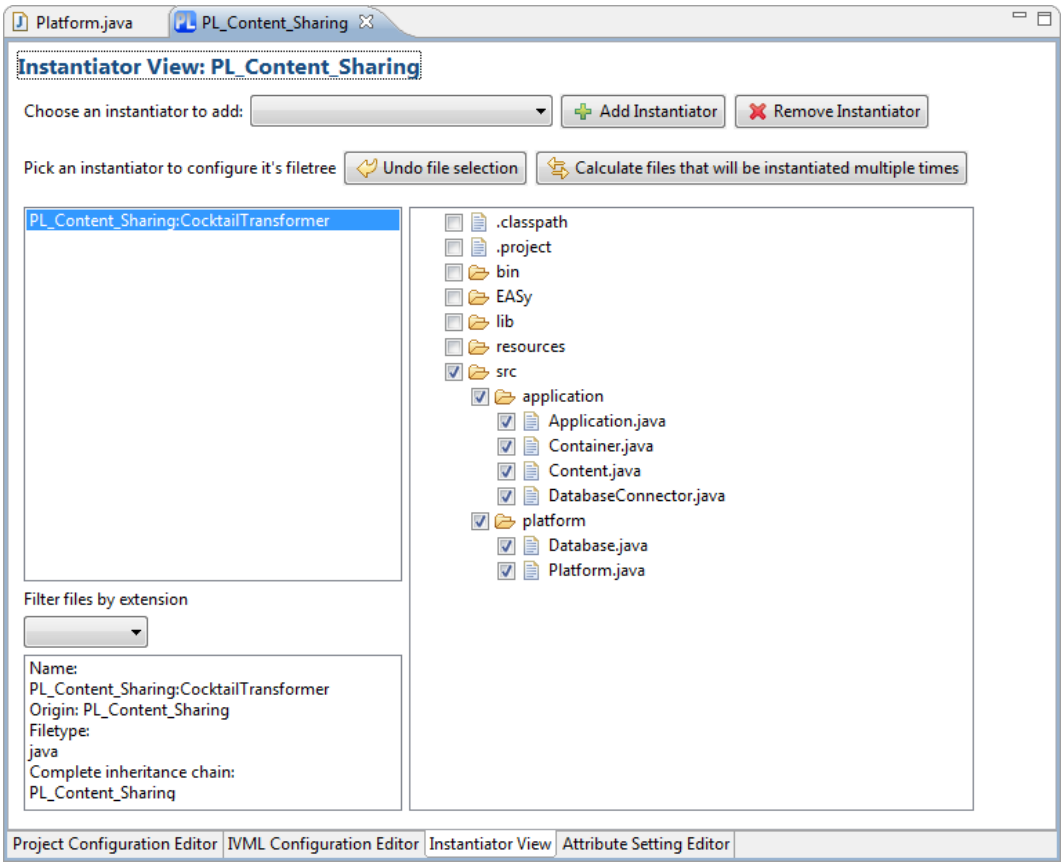

Figure 5: Defining an instantiator for artefact instantiation.

The last step is to define how the generic artefacts defined for the variant-enabled base platform will be turned into artefacts of the instantiated domain-specific platform. Currently, EASy-Producer offers two ways: 1) to instantiate the artefacts in terms of their structure, i.e., by taking over existing namespaces, packages and class / file names; 2) to add additional namespaces for distinguishing the artefacts combined from different product lines.

The creation of namespaces as part of deriving instantiated product line artefacts is related to the concept of Multi-Software Product Lines (MSPL). In MSPL scenarios, a product inherits the functionalities of more than one product line in terms of including all artefacts of all (parent-) product lines with respect to the products configuration. EASy-Producer provides MSPL capabilities and, thus, allows adding or removing predecessors (i.e. additional parent-product lines) for a product line project. A predecessor in EASy-Producer is a product line project from which the derived project (product line or product) will inherit its functionalities. For example, the application and the platform of the content-sharing application can be implemented as two different product line projects. A product which integrates both functionalities (like the audio content-sharing application of the running example) can be derived from one of the product line projects, while the other product line can be added by clicking the *Add/remove predecessor* button in the **IVML Configuration Editor** tab. Clicking the *Pull Configuration* button in this tab will then refresh the configuration options in terms of adding the new options from the new predecessor to the editor. The configuration of all (integrated) configuration options

will yield a new product, which combines the functionality of both predecessors according to the configuration.

However, in our example, we will use the first (simple) mechanism relying on the current artefact structure. Thus, switch to the **Project Configuration Editor** tab and use the drop-down menu to select the *Copy without package adaptation* entry as shown in Figure 6.

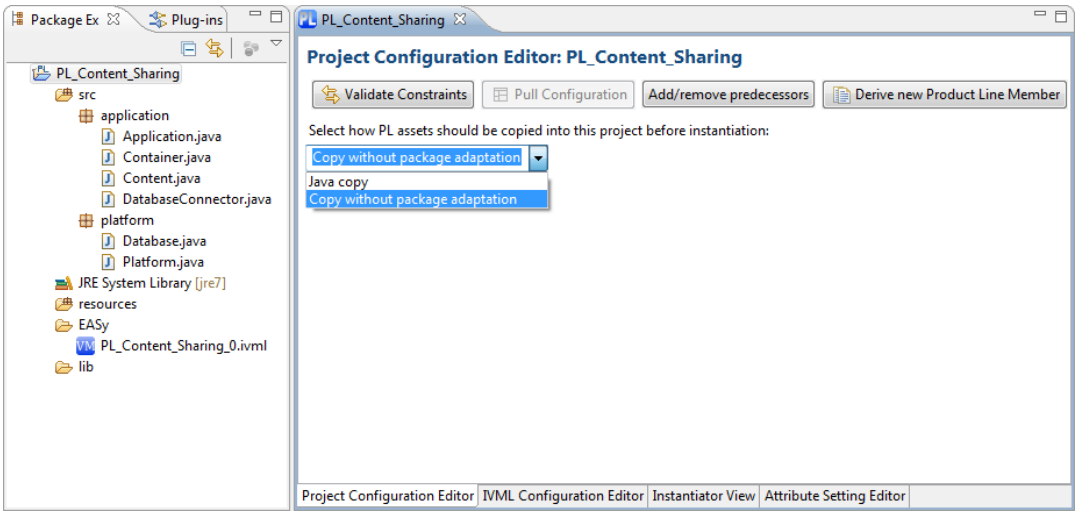

Figure 6: Selecting the copy-mechanism for artefact derivation.

Finally, the implementation space is defined and an instantiator is assigned to instantiate content-sharing application variants accordingly to a configuration. On this basis, we will derive a new product from this product line in the next section.

2.2 Deriving a Domain-Specific Service Platform

In this section, we will describe the process of deriving a new domain-specific platform from a software product line defined in EASy-Producer. We will adopt the perspective of a Platform Variant Creator and start with the derivation of a new product line member[4] (in this case, the product project), configure the product based on the variability model defined in Section 2.1.1, and instantiate the product line artefacts accordingly. This will result in a specific content-sharing application variant with the desired functionalities ready for use.

The first step towards an instantiated domain-specific platform is to derive a new member from the previously defined base platform product line. For this purpose open the **Product Line Editor** by right clicking on the *product line project* and select *Edit Productline* in the context menu. In the **ProjectConfiguration Editor** tab click the *Derive new Product Line Member* button, define a name for the new member, and click the *Ok* button. In our running example, we will use `Audio_Sharing_App` as the name of the new member. A new product line project will be created and the corresponding **Product Line Editor** will open automatically as shown in Figure 7.

[4] In EASy-Producer, we do not distinguish between a product line infrastructure and a final product. Both are simply projects that may contain more or less variability (in case of a product none)

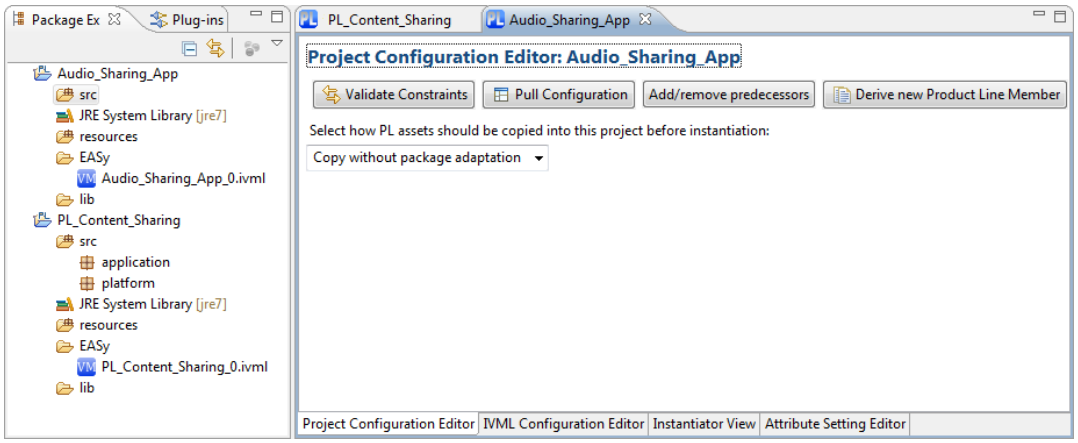

Figure 7: New product line member derived from a product line.

In the new product line member project, we will configure the desired functionalities of our specific audio content-sharing platform. This configuration will be used to finally instantiate the domain-specific platform. We will describe both steps in detail in the next two sections.

2.2.1 Configuration of a Domain-Specific Service Platform

A product configuration (in this example the configuration of the domain-specific service platform) is a set of configured elements. In IVML configured elements are specified by assigning specific values to the elements in the configuration space, i.e. the decision variables, the attributes, etc. The validity of a configuration is checked against the constraints of the variability model using the built-in reasoning mechanism. The valid product configuration provides the basis for the (automated) instantiation of the corresponding product artefacts.

EASy-Producer provides two ways of configuring the elements of an IVML variability model: either use the **IVML Editor** by double-clicking the *IVML file* of the derived product line member (in our example the `Audio_Sharing_App_0.ivml` file) in order to configure the elements of the imported project (the product line project) manually, or use the **IVML Configuration Editor** tab of the **Product Line Editor**. In our example, we will use the **IVML Configuration Editor**. This eases the configuration task as it includes all configurable elements of the imported project and provides the possible values for each of these elements automatically (we will discuss the configuration editor in detail in Section 3.5). Figure 8 illustrates the **IVML Configuration Editor,** including the configurable elements of our audio content-sharing application.

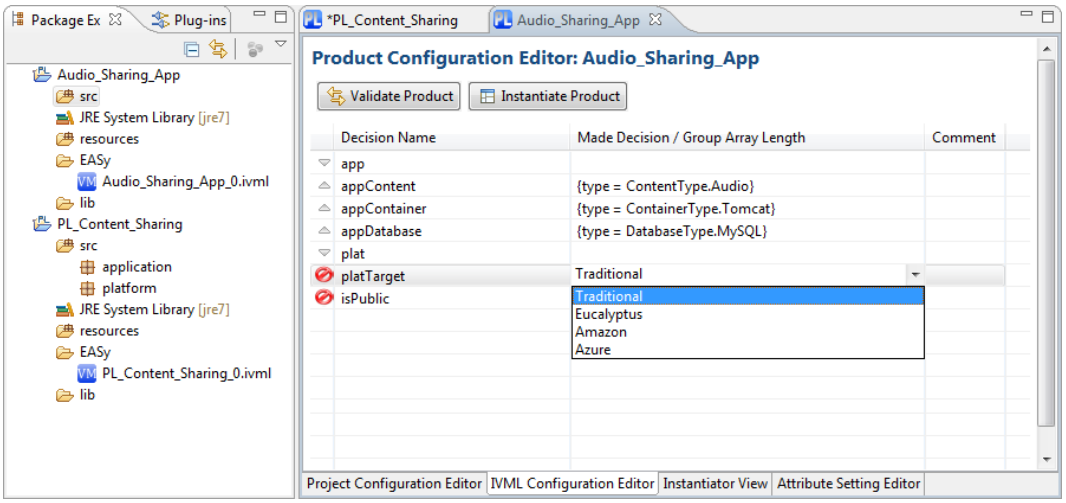

Figure 8: Product configuration using the IVML configuration editor.

The next step is to check whether the configuration is valid. For this purpose, click on the *Validate Product* button of the **IVML Configuration Editor**. This executes the built-in IVML reasoning. If the product is valid, it is ready for instantiation. If it is not valid, the configuration must be revised in order to guarantee that the resulting product will work appropriately. In case of an invalid configuration, EASy-Producer will issue a description of the configuration problem and propose a possible error location in the current configuration. Please not that this kind of user support is still under development and may not work appropriately in every situation in the current release of EASy-Producer. However, we are working on this kind of support as part of future releases to ensure scalability to large and complex configuration problems.

Finally, the product is configured and ready for instantiation.

2.2.2 Instantiation of a Domain-Specific Service Platform

Product instantiation describes the process of resolving the variability of product line artefacts according to a product configuration. This process results in the product artefacts that are mostly variation-free and ready to use. However, in some situations it is desired to resolve some of the variabilities at a later point in time, for example, at initialization time or runtime. In such a case, the instantiation process will leave these variabilities as-is.

EASy-Producer provides a fully automated instantiation process, which is based on the variability model, the current configuration and the selected instantiators. We defined this information in the previous sections, such as the implementation space and instantiator definition (cf. Section 2.1.2) and the product configuration (cf. Section 2.2.1). This relies in turn on the configuration space definition (cf. Section 2.1.1). Thus, the only remaining activity is to click the *Instantiate Product* button in the **IVML Configuration Editor**. This will yield the instantiated artefacts from the product line project and insert them into the product project by resolving the variabilities.

3 Variability Engineering Tool Design

In this section, we will discuss the design of the Variability engineering tool EASy-Producer as well as the state of the realization of the main components. In Section 3.1, we will introduce the overall architecture of EASy-Producer. In the following sections we will discuss particular components of the architecture in more detail: in Section 3.2 the realization of the INDENICA Variability Modelling Language (IVML), in Section 3.3 the reasoning support for IVML, in Section 3.4 the product instantiation support, and in Section 3.5 the specific support for the application engineer via the user interface.

3.1 Architecture

In this section, we describe the architecture of EASy-Producer. First, we will discuss the main design principles and then the architecture in terms of individual (logical) components. Finally, we will outline the realization and validation status of the entire EASy-Producer tool.

The design of EASy-Producer follows three main design principles:

- **Separation of functionality:** The core of EASy-Producer consists of commonly used functionality such as management of variability models or product line projects. Specific functionality such as individual instantiation mechanisms or even reasoners is separated from the core. These components extend the core, i.e., they provide specific functionality, which is called by the core at certain points during the execution.
- **Separation of the user interface:** Functional components such as the variability instantiation mechanisms do not provide a user interface. This facilitates the direct reuse of the functional components in headless product build workflows or in the INDENICA runtime environment (WP4). However, the user interface relies on the underlying functional components.
- **Separation of the external tool integration:** The components, which realize the integration with other INDENICA tools rely on core functionality and use specific functionality only through the core. The integration components are separated into specific interfaces to be used by the external tools as well as their actual implementation. Integrations shall be optional for the external tool, i.e., it shall be able to work also without an installed version of EASy. Thus, the integration in the external tool only becomes active when also their counterpart in terms of EASy components is installed. However, in future integration components may also contribute to the user interface and, thus, would further be split into a user interface and a functional part as stated above.

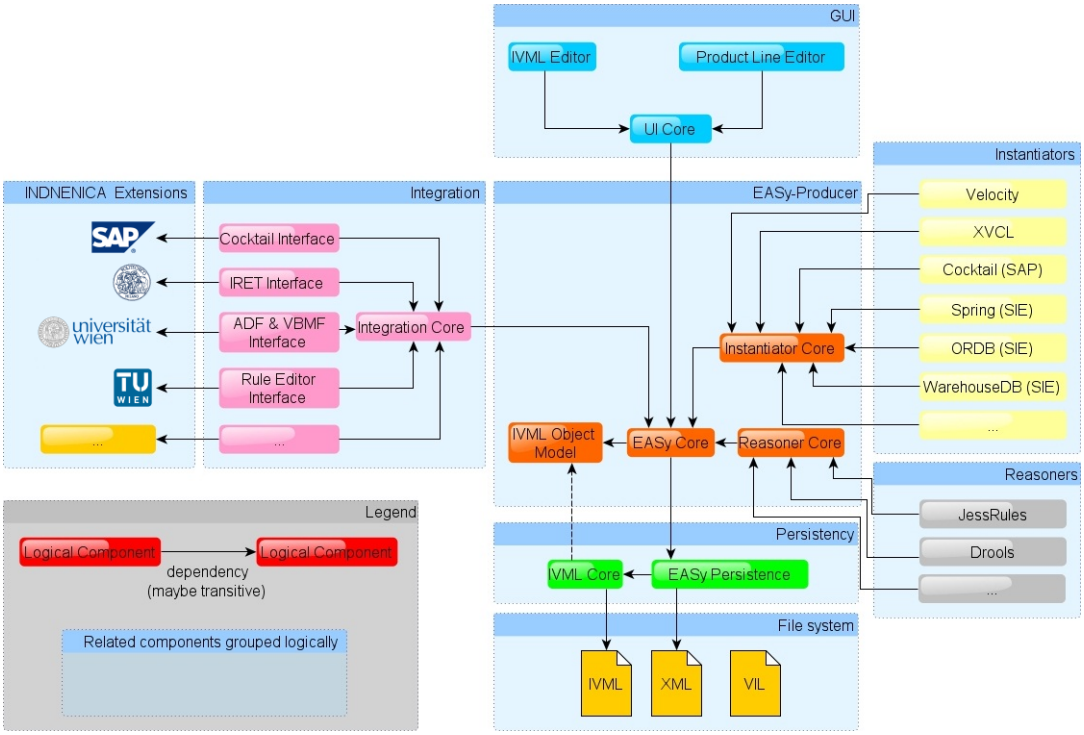

Figure 9: Architecture of the Variability Engineering tool „EASy-Producer“

Following these three principles, we designed the architecture of EASy-Producer as illustrated in Figure 9. The architecture defines a number of components, each with provided and required interfaces. Components, which have (partly) been taken over from previous versions of EASy-Producer are marked below accordingly.

- **EASy-Producer core:** The functional core of the EASy-Producer tool consists of five components:
  - The ***IVML object model*** realizes an object-oriented representation of IVML variability models. The object model provides classes, which realize individual IVML concepts, a tree-based representation of IVML constraints as well as model management functionality supporting model imports and compositions.
  - The ***reasoner core*** manages available reasoner components and provides access to high-level reasoning functionality such as model consistency checks, value propagation and constraint evaluation. Multiple reasoners can be plugged into the reasoner core, which, in turn, serves as a basis for selecting the most appropriate reasoner for a given IVML model. Currently, two specific reasoners are realized as depicted in Figure 9, but further ones are under realization or planned.
  - The ***instantiator core*** manages available variability implementation techniques realized as individual instantiator components. The current implementation of the instantiator core is based on the previous version of EASy-Producer, while future versions of this component will support the execution of a descriptive specification of the production workflow in VIL (see D2.2.1, Section 5 for an initial description; the detailed concepts will be presented in D2.2.2).

  - The **IVML core** contains a parser as well as a semantic analyser for IVML models and allows obtaining a (valid) IVML object model from textual input. IVML core and IVML object model interact in particular during model import and composition, i.e., the model management of the IVML object model determines which models to load based on IVML import statements and the IVML core creates the related object models on demand. Further, persistency mechanisms provide access to product line specific information such as predecessors in a hierarchical multi-software product line.
  - The **EASy core** realizes functionality for hierarchical product line engineering on top of the components described above. The base implementation of this component has been taken over from the previous version of EASy-Producer and was adapted to the current architecture as well as to the new concepts introduced by the IVML.
- **Reasoners:** A reasoner component realizes the translation of IVML concepts to reasoner-specific concepts as well as the translation of reasoner results to an IVML configuration. A reasoner component is an extension of the reasoner core. Currently, we realized two reasoner integrations (see Section 3.3).
- **Instantiators:** A specific instantiation component realizes one or multiple variability implementation techniques. Two generic instantiators, namely the Velocity- and XVCL-instantiator have been taken over from the previous version of EASy-Producer. INDENICA-specific instantiators have been realized to support specific variability implementation techniques that were relevant to the development technologies used by the different partners. The previously mentioned SAP Cocktail instantiator (see Section 2.1.2) is one such example; further ones will be realized / refined in the remainder of the project (see Section 3.4).
- **User interface:** The user interface integrates EASy into the Eclipse IDE. Currently, the user interface contributions of EASy consist of the syntax-driven editor for IVML models (see Section 2.1.2), a table-based editor for supporting the configuration tasks of an application engineer (see Sections 2.2.1 and 3.5) as well as several configuration dialogs. An editor for simplifying the variability modelling with IVML is scheduled as future work. Basically, the individual editors are optional. For example, one version of EASy-Producer may rely just on the simple table-based editors, thus hiding the advanced modelling concepts provided by IVML, while other versions may also include the IVML editor. Further, the entire user interface is optional to facilitate the application of EASy in headless settings such as build mechanisms or the INDENICA runtime environment (WP4).
- **Tool integrations:** EASy-Producer integrates with other tools developed in INDENICA (see also Section 4). For the integration with the other INDENICA development time tools, EASy-Producer offers interfaces to the INDENICA requirements engineering tool IRET (see also Deliverable D1.2.1), to the architecture decision framework ADF (Deliverable D1.3.1) and to the view-based modelling framework VBMF (Deliverables D3.1 and D3.3.1) as well as to the WP4 monitoring and adaptation rule editors (Rule Editor integration component). Certain integrations will also interact with the runtime

environment. Some integration will be realized through artefact instantiation, like deriving instantiated monitoring and adaptation rules, while others will happen through connectors to instantiated systems (Cocktail integration component). We will detail the integration with other WPs in Section 4.

All components described above have been realized as individual OSGi[5] components (bundles). We selected OSGi as it is the underlying technology for the Eclipse IDE and serves as common platform for the INDENICA development time tools. The extending components such as the reasoner integrations or the instantiators are realized as OSGi declarative services[5]. While the user interface components of EASy are Eclipse plug-ins, the remaining (functional) components are pure OSGi components in order to make EASy available to headless build workflows as well as for integration into the WP4 runtime environment. An Eclipse update site has been set up[6] which provides access to all realized components and facilitates user-friendly installation into the Eclipse IDE.

The integrated EASy-Producer tool will be subject to an extensive regression test suite in the future. However, the logical components are already tested individually, while we validate the integrated installation of EASy components manually before new versions are released to the EASy update site. Currently, we provide bi-weekly releases of the EASy components. In addition to the implementation, we provide documentation in terms of two life documents, a user guide[7] and a developer guide[8].

3.2 INDENICA Variability Modelling Language (IVML)

The INDENICA Variability Modelling Language (IVML) is a novel language for textual variability modelling. IVML is explicitly based on the project requirements as discussed in D2.1 and we found no other existing approach that would adequately address this wealth of requirements. The approach was designed with the whole breadth of requirements in mind that are relevant to variability in platform customization and integration. In this section, we provide an overview of the current state of the IVML implementation based on the concepts in D2.1 as well as its validation.

We developed a faithful implementation of the IVML language concepts defined in D2.1 as part of the realization of the WP2 variability engineering tool. However minor (syntactical) inconsistencies in the language design have been detected and corrected as part of the implementation work. We maintain a life document, the IMVL language specification[9], as documentation of the current state of the IVML language, its concepts and its semantics. This document also contains a detailed description of the IVML constraint language, the operations available in constraints as well as the entire grammar of IVML.

[5] http://www.osgi.org/Specifications/HomePage

[6] EASy-Producer Update Site: http://projects.sse.uni-hildesheim.de/easy/

[7] EASy-Producer User Guide: http://projects.sse.uni-hildesheim.de/easy/docs/guide.pdf

[8] EASy-Producer Developers Guide: http://projects.sse.uni-hildesheim.de/easy/docs/devguide.pdf

[9] IVML language specification: http://projects.sse.uni-hildesheim.de/easy/docs/ivml_spec.pdf

We implemented IVML in terms of three EASy-components: the IVML core, the IVML editor and the IVML object model. Large parts of the IVML core and the IVML editor have been realized using the Xtext[10] software development kit. Xtext is a popular open-source technology for developing editors and language-related tools for textual (domain-specific) languages for the Eclipse IDE. Based on a given grammar, Xtext generates a parser as well as parts of the user interface such as a (customizable) syntax-driven editor with syntax and problem highlighting, an interactive content assist mechanism and a hierarchical outline. Xtext also supports the development of the semantic analysis in terms of a customizable type system and (incremental) semantic validation. However, we decided to realize the semantic analysis for IVML manually in order to have explicit control over the functionality and to avoid non-trivial incompatibilities with evolving versions of Xtext and to ease the realization of headless versions of EASy-Producer such as the WP4 integration. We also realized the IVML object model manually. Here, an option would have been to use the Ecore-based abstract syntax representation generated by Xtext as a basis, but this would imply a mix of syntactical representation on grammar level with language concepts. The result of a successful semantic analysis of a valid IVML model is an instance of the IVML object model, which is the basis for realizing higher-level functionality in EASy. All concepts of IVML, as well as their semantics, have been realized at this point. Further, we customized the generated IVML editor by an IVML-specific outline. A supporting content-assist mechanism will also follow in one of the next releases.

The IVML components, in particular the parser implementation, the semantic analysis as well as the IVML object model are subject to an extensive regression test suite. The IVML object model is validated by 97 unit test cases. Parser and semantic analysis are validated by 53 test cases given in terms of individual IVML models, which cover all IVML concepts. Further, we use the most current versions of the variability models of the INDENICA partners as test cases.

3.3 Reasoning support for IVML

The IVML language provides highly expressive modelling elements and concepts for the definition of variability models. Thus, checking whether a specific (product) configuration is valid is a challenging task. In EASy-Producer, we use so-called reasoners to perform the task of model and configuration checking and validation. A reasoner is typically a third-party tool, which is designed to solve logical and combinatorial problems, checking specific value combinations of related modelling elements, etc. In this section, we will discuss the reasoning support for IVML. This will include the description of the approach to select appropriate reasoners for IVML as well as the current state of implementation of the selected reasoners.

While the issue was raised that some IVML models may yield undecidable logical and combinatorial problems and, thus, may not be reasonable at all, we can exclude such scenarios in the particular context of the INDENICA project. A running platform needs to be fully defined in order to be executable at runtime and provide the

[10] Xtext website: http://www.eclipse.org/Xtext/

desired functionalities. At runtime, the expressions of the IVML language only have to be evaluated based on a rule-driven mode: thus for each evaluation at least one additional value is set with the total set of values being limited. With the maximum number of the values being limited to at most several thousands (typically more in the hundreds), we do not expect any performance issues (see initial experimental results in Section 3.3.2). Further, for fully defined value sets, the problem degenerates to checking ground instances, which is trivially decidable in our language. We expect that in realistic scenarios fewer than 10% of the variable values need to be deduced at runtime, leading to values of a few dozen variables. At development time, we aim at providing reasoning capabilities for supporting the application engineer by trying to set any implicitly defined values automatically. One should note, however, that this is purely a comfort function. This will be done in a way that ensures that also intermediate deductions are fully correct and that this assistance functionality is non-blocking. This will avoid any negative effects from high running times (these can also be expected to be highly unlikely for realistic situations according to theoretical analysis from the area of product lines [4]).

3.3.1 Reasoner Selection

In order to find appropriate reasoners, we performed an extensive analysis of existing solvers, model checkers, etc. We analysed 27 different tools in total. However, we will only discuss those here, which we analysed in detail, as some of the reasoners could be immediately excluded, e.g., due to incomplete or no documentation, outdated releases, etc. The criteria for the detailed analysis were derived from the concepts of the IVML language as we aimed at reducing effort on implementing IVML reasoners by reusing existing capabilities and, if required, adapt them to process IVML modelling elements correctly. We applied the following categories of criteria:

- **Type support:** A reasoner must support at least all basic types of the IVML language. This includes *Boolean* as well as *non-Boolean* types, such as Integer, Real, and String. It shall also support *enumerations*, but this may also be simulated by Integers. Further, advanced types like *containers* (set and sequences) and *compounds* shall be supported. However, also compounds may be simulated, if not supported directly, but this increases the translation and development effort. Further, we do not specifically require the support of *derived types* as they can be simulated in terms of basic types and constraints.
- **Constraint support:** A reasoner must support different types of operators in order to define constraints among IVML variables of the above types. This includes *logical* and *arithmetic* operators as well as operators for *set and sequences*, for example quantification operators. Arithmetic operators also lead to the requirement of being able to *compute* derived values.
- **Configuration support:** In general, a reasoner shall support model consistency checking. Further, it shall support (mechanisms to determine) *undefined values* and *value propagation* (derivation), *partial configuration* (this includes support for undefined values) as well as *partial* e*valuation*.

These mechanisms are required to facilitate multi-staged configuration within IVML and EASy-Producer.

- **General:** We also include some meta-criteria in order to select appropriate reasoners for IVML and, in particular, for the INDENICA project. The *implementation* of a reasoner shall be realized in Java (or at least provide a Java native interface) for an easy integration with EASy-Producer and with other tools developed in INDENICA. However, we would also select reasoners of any other implementation, if they provide a better support than reasoners implemented in Java. In case that a potential reasoner may also offer *extensibility* capabilities (i.e., an API and a corresponding *license* agreement, which allow extending the base capabilities), we will prefer such a reasoner as this will ease the implementation of additional operations.

We included a large set of solvers, model checkers, etc. of different categories in our analysis, namely Constraint Satisfaction Problem (CSP) solvers, Object Constraint Language (OCL) reasoners, Description Logic (DL) solvers, Satisfiability Modulo Theorem (SMT) provers, and rule engines. We opted primarily for the inclusion of rule engines during our analysis as we realized that existing reasoners did not properly support our requirements, while rule engines offer a vast set of capabilities and functionalities required. Further, we excluded SAT-solvers as these solvers do not support non-Boolean types and expressions. Table 1 summarizes the results of our analysis.

In the category of CSP solvers, we analysed Gecode[11], Choco[12], and the OR-Tools[13]. In general, these solvers provide good support for Boolean and Integer types. Thus, these solvers also provide logical and arithmetic operators for defining constraints among variables of these types. However, in these tools the definition of Integer variables requires the definition of a fixed range of possible values, while in IVML variables of this type are in general unbounded (if no constraint restricts the set of possible values). Gecode and Choco also provide a real type, however, variables of this type have to be bound to a fixed range of values as well. Further, most analysed tools do not support basic types like strings and enumerations at all. More complex types like containers are only supported by Gecode and Choco as bounded Integer sets. Thus, operators for constraining such sets can only be applied to sets of type Integer. Compounds are not supported at all. Regarding configuration support, these solvers allow partial configuration, i.e., defining specific values for a subset of all available variables, while undefined values will be derived, for example by evaluating the constraints. However, this is of course limited to the supported types. In the category of general criteria, all reasoners provide extensibility under the terms of less restrictive licenses. However, as a result, we decided that CSP solvers would not be the first choice for IVML reasoning as their type system as well as their constraint system is too limited and will not be sufficient to translate all IVML elements and

[11] Gecode website: http://www.gecode.org/

[12] Choco website: http://www.emn.fr/z-info/choco-solver/

[13] OR-Tools website: http://code.google.com/p/or-tools/

Criteria		CSP			OCL		DL				SMT				Rule engines	
		Gecode	Choco	OR-Tools	EMF-OCL	Dresden OCL	Pellet	Racer	FaCT	HermiT	Alt-Ergo	CVC3	Yices	Z3	Jess	Drools
Type support	Boolean	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Non-Boolean	o	o	o	o	o	o	o	o	o	o	o	o	o	x	x
	Containers	o	o	o	x	o	x	x	x	x	o	x	o	o	x	x
	Compounds	-	-	-	x	o	x	x	x	x	-	x	o	o	x	x
Constraint support	Logical operators	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
	Arithmetic operators	x	x	x	x	x	o	o	o	o	x	x	x	x	x	x
	Set and sequence operators	o	o	o	-	-	o	o	o	o	o	o	o	o	x	x
Configuration support	Undefined values	o	o	-	-	-	-	-	-	-	o	o	o	o	-	-
	Value propagation	o	o	-	-	-	-	-	-	-	o	o	o	o	-	-
	Partial configuration	o	o	-	-	-	-	-	-	-	x	x	x	x	-	-
	Partial evaluation	-	-	-	-	-	-	-	-	-	x	x	x	x	o	o
General	Implementation	C++	Java	C++	Java	Java			Lisp	Java	OCaml	C/C++			Java	Java
	Extensibility	x	x	x	-	-	-	-	-	x	-	-	-	x	X	x
	Licensing	MIT	BSD	Apache License 2.0	EPL	EPL	proprietary	proprietary	GNU GPL	GNU GPL	CeCILL-C	BSD	proprietary	proprietary	Reasearch: free Commercial: proprietary	ASL 2.0

Table 1: Reasoners analysis results

concepts to their models. Further, the extension of such tools would also be limited due to the core capabilities of these reasoners.

The OCL libraries EMF-OCL[14] and Dresden OCL[15] were promising candidates for IVML reasoning as the IVML constraint language is based on OCL, i.e., the type system and most operators are appropriate while default values as well as value assignments as required for defining configurations need to be added. However, these tools only perform pure model checking and cannot resolve dependencies and relations between constraints and value definitions.

In the category of DL solvers, we analysed Pellet OWL 2 Reasoner[16], Racer Pro[17], FaCT[18], and HermiT[19]. While all of these tools are able to fulfil the requirements drawn by the IVML type system, their constraint languages are rather limited [3]. In general, these tools only support a subset of operators for set and sequences provided by IVML. Further, these tools typically do not support value calculation and require values (called facts) to check whether a model is valid. Thus, these tools do not directly support partial configuration and evaluation. However, the most crucial reason for excluding DL solvers was that most of these tools are only available under proprietary licenses and, thus, cannot be freely used in the project and extended in order to implement missing functionalities.

[14] EMF-OCL website: http://www.eclipse.org/modeling/mdt/downloads/?project=ocl

[15] Dresden OLC website: http://www.dresden-ocl.org/index.php/DresdenOCL

[16] Pellet website: http://clarkparsia.com/pellet/

[17] Racer Pro website: http://www.racer-systems.com

[18] FaCT website: http://www.cs.man.ac.uk/~horrocks/FaCT/

[19] HermiT website: http://www.hermit-reasoner.com/

In the category of SMT solvers, we analysed Alt-Ergo[20], CVC3[21], Yices[22] and Z3[23]. These solvers also provide good support for the IVML type system, however, only some (CVC3 and Z3) provide concepts similar to containers and compounds as required by IVML. While these tools also provide the full set of required Boolean and arithmetic operators (including value calculation), set and sequence operators are limited to a subset of IVML container operations. Also these solvers require values (called asserts) to check whether a model is valid, which also leads to unsupported partial configuration and evaluation. Again, extending these tools in order to implement these functionalities is not possible due to their implementation and licensing. Although the basic reasoning capabilities of this category may serve as a good basis, we excluded them initially due to the limited set of operators in combination with the restricted opportunities for adding required operations.

The final category is rule engines. In this category, we analysed Drools[24] and Jess[25]. Both engines support the complete type system as well as the constraint system (including value calculation) of the IVML language. These engines also support partial evaluation based on asserted facts. However, undefined values as well as partial configuration are not supported per se but can be implemented in terms of user-defined functions. Further, both engines provide for extensibility of the existing set of operations by user-defined functions and by functionality implemented as (external) Java functions.

The result of our analysis of existing solvers, model checkers, etc. turned out that rule engines are a solid basis for the IVML reasoning support. While these engines also have certain disadvantages in terms of not supporting undefined values and partial configurations, and value propagation, they provide the largest set of types and operations for an easy translation between IVML concepts and the concepts of the analysed engines. Further, their extensibility allows defining additional functionality and, thus, simulating IVML concepts not supported in general. This will reduce the effort for implementing these engines as IVML reasoners, which in turn will offer the opportunity to develop additional concepts to eliminate the disadvantages in the future, e.g., by realizing the missing functionality in terms of a combination of reasoners.

3.3.2 Reasoner Implementation

Currently, reasoning for IVML models is being implemented in terms of two alternative reasoner components, one using the rule engine Jess and another based on JBoss Drools. The reasons for selecting rule engines were described in Section 3.3.1. In this section, we discuss why we realized two reasoning components, their implementation status as well as initial performance experiments.

[20] Alt-Ergo website: http://alt-ergo.lri.fr/

[21] CVC3 website: http://www.cs.nyu.edu/acsys/cvc3/

[22] Yices website: http://yices.csl.sri.com/

[23] Z3 website: http://research.microsoft.com/en-us/um/redmond/projects/z3/

[24] Drools website: http://www.jboss.org/drools/

[25] Jess website: http://www.jessrules.com/

Priority	Target	No.	Implementation status					
			Jess			Drools		
			Implemented	Partially implemented	Not implemented	Implemented	Partially implemented	Not implemented
1	Concepts	25	23 (**92%**)	1 (**4%**)	1 (**4%**)	23 (**92%**)	1 (**4%**)	1 (**4%**)
	Operations	72	72 (**100%**)		-	72 (**100%**)		-
2	Concepts	2	1 (**50 %**)	-	1 (**50%**)	1 (**50 %**)	-	1 (**50%**)
	Operations	9	9 (**100%**)		-	9 (**100%**)		-
3	Concepts	12	5 (**42%**)	6 (**50%**)	1 (**8%**)	5 (**42%**)	6 (**50%**)	1 (**8%**)
	Operations	61	48 (**79%**)		13 (**21%**)	38 (**62%**)		23 (**38%**)
4	Concepts	6	-	-	6 (**100%**)	-	-	6 (**100%**)
	Operations	6	-		6 (**100%**)	-		6 (**100%**)
Total:	*Concepts*	*45*	*29 (**64%**)*	*7 (**16%**)*	*9 (**20%**)*	*29 (**64%**)*	*7 (**16%**)*	*9 (**20%**)*
	Operations	*148*	*129 (**87%**)*		*19 (**13%**)*	*119 (**80%**)*		*29 (**20%**)*

Table 2: Implementation status of the reasoning components.

3.3.2.1 Implementation Status

Our initial implementation of the IVML reasoner was based on Jess rule engine. This is attributed to the efficiency of Jess and wide applicability, as observed in [2, 5]. However, due to practical issues identified during the realization of the Jess reasoning component (we will discuss these issues in detail below) we also realized an alternative reasoning component based on Drools expert, an open-source component provided by the JBoss project. Drools has also been adopted extensively in knowledge-based expert systems, for example in [1, 6]. The Drools-based implementation also solves some practical issues, e.g., relating to licensing as identified with the Jess-based component.

We grouped the IVML concepts as well as the related constraint operations defined in D2.1 into four priority levels in order to structure and focus the implementation. We derived these priority levels from the usage of IVML concepts in the models we obtained from our industrial partners. The priority levels are:

- Priority 1 (highest): Decision variable definitions for basic types (excluding String) and compounds as well as constraints and constraint operations involving these basic types and compounds.
- Priority 2: Decision variable definitions for strings, constraints on strings as well as types derived from al basic types or compounds including related constraints and constraint operations.
- Priority 3: Decision variable definitions for containers such as sets and sequences as well as constraints and constraint operations on containers.
- Priority 4 (lowest): Advanced constraints containing ‘let expressions’ or ‘if-then-else’.

Table 2 summarizes the implementation status of the reasoning components in terms of the number of (partially) implemented or currently not implemented IVML

concepts and constraint operations for each priority level[26]. Table 2 shows that a significant amount of implementation has been achieved for the concepts and operations[27] that are flagged with a high priority (1-3) and work is underway to realize priority 4 concepts and operations. In summary, we classified the IVML concepts into 45 constructs, of which 29 (64%) are realized completely, 7 (16%) are realized partially, and 9 (20%) are currently not yet realized for both Jess and Drools. Further, 129 (87%) operations are implemented for Jess while 119 (80%) are implemented for Drools. In essence, the reasoning components currently realize 80% of the functionality required by IVML.

The implemented concepts and operations are subjects to an extensive test suite, which can be generically applied to arbitrary reasoner components, which implement the interfaces provided by the reasoner core. Currently, this test suite consists of 38 test cases in total, while a single test typically addresses multiple similar operations, e.g., arithmetic operations are validated by a single test case.

Although the Jess component offers a good starting point to reason over IVML variability models and configurations, it poses certain practical limitations. To mitigate these limitations, we performed a detailed comparison of the concepts provided by Jess and Drools. Based on this comparison and our experience made during the development of the Jess component we were able to quickly provide an alternative reasoning component based on Drools. The practical limitations of Jess as well as the mitigations we developed based on Drools are discussed below.

- **Side effects in Jess pattern matching may cause infinite loops**: At the core, the Jess rule engine is based on pattern matching over facts in memory. We encountered a few scenarios where the rule engine runs into an endless loop. To mitigate such problems, we developed a mechanism which terminates Jess according to a given time constraint. Thus, in case of continuous loop events, the termination of the reasoning process is controlled from outside the actual reasoning engine. However, currently this will result in not having any reasoning results, but EASy Producer will inform the user that such a loop exists in the current model. In contrast, Drools comes with a mechanism, which stops the immediate firing of the rule that modifies a certain variable in the working memory, thus avoiding infinite loops.
- **Restriction of the commercial use due to the Jess license**: The vendors of the Jess rule engine provide their implementation using a dual license model: for academic research the use of Jess is free, while a commercial license must be purchased for use in industry. Drools overcomes this limitation completely as it is available based on the Apache Software License, which allows both academic and commercial usage.
- **Limitation over reasoning on multiple instances of a single compound type**: We identified that reasoning over constraints, which relate multiple instances

[26] Please note that partial implementation is not applicable to operations; they are either completely implemented or not implemented at all).

[27] Please note that the total number of operations in the IVML specification for a given concept is different from the number illustrated in Table 2. This is due to implementation reasons such as individual operation signatures for overloaded operators.

Number of Elements		Jess						Drools								
Vars.	Constr.	Translation time			Reasoning time			Translation Time			Parsing Time			Reasoning Time		
		Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
Complexity Level 1																
10	1	0.020	0.032	0.023	0.012	0.013	0.124	0.002	0.003	0.003	0.780	0.891	0.883	0.887	0.901	0.892
100	10	0.055	0.057	0.056	0.030	0.036	0.032	0.004	0.006	0.004	1.040	1.080	1.060	1.059	1.09	1.067
1000	100	0.039	0.418	0.046	0.320	0.339	0.329	0.051	0.056	0.053	4.080	4.230	4.120	4.120	4.270	4.160
Complexity Level 2																
10	1	0.002	0.022	0.021	0.013	0.014	0.013	0.002	0.003	0.003	0.930	0.950	0.940	0.94	0.96	0.952
100	10	0.056	0.058	0.057	0.033	0.039	0.035	0.004	0.005	0.004	1.320	1.370	1.330	1.33	1.38	1.343
1000	100	0.402	0.450	0.413	0.433	0.450	0.440	0.059	0.063	0.060	7.020	7.270	7.170	7.080	7.330	7.230

Table 3: Reasoning performance (given in seconds).

of a compound type, implies serious restrictions in Jess. Jess does not validate such constraints because it evaluates individual rules based on the type of the used elements rather than based on individual instances. However, Drools does not impose such restrictions and, thus, supports reasoning on multiple instances of a single compound type.

- **No support for value propagation:** Value propagation enables the inference of values based on the restrictions implied by the constraints. For example, multiple constraints may restrict the value range from Integer to exactly one specific value. In particular, both reasoners only support reasoning over variables that have been assigned with specific values. However, there is a Drools extension called Drools Planner[28], which offers planning-capabilities in terms of optimization algorithms and construction heuristics. We are currently investigating this opportunity to enable value propagation over IVML projects.
- **Validation of IVML-models:** Jess and Drools do not support the validation of pure IVML-models, i.e., checking whether at least one configuration can be derived from an IVML model. This limitation is closely related to the unsupported value propagation discussed previously. Thus, we expect the Drools Planner tool to be a possible solution.

In summary, both reasoners provide good reasoning capabilities for IVML. The implementation of the complete set of concepts and operations as well as the solution to the open issues will be part of the realization efforts of year 3. While some issues and limitations exist, Jess and Drools performed very well for each configuration given so far in INDENICA. However, we will discuss the results of an initial performance experiment in the next section.

3.3.2.2 Initial Performance Results

We performed an initial set of scalability tests in order to evaluate the performance of both Jess and Drools. For this purpose, we automatically generated IVML models of different complexity and with different numbers of elements:

- **Number of elements:** We started with 10 variables of Boolean and non-Boolean types and 1 constraint while increasing these numbers

[28] Drools Planner website: http://www.jboss.org/drools/drools-planner

simultaneously by a factor of 10. The largest IVML file contained 1000 variables and 100 constraints. We selected these numbers as the ratio of variables and constraints typically is 10:1 in large-scale, real-world variability models, i.e. as described in [7].

- **Complexity Level:** The complexity levels describe the complexity of the generated constraints. Constraints of complexity level 1 will be simple constraints, e.g., $a \text{ and } b$ (a and b being Boolean variables) or $c > d$ (c and d being Integer or real variables). Constraints of complexity level 2 will concatenate three simple constraints by Boolean operators and negation. Further, on complexity level 3, constraints may contain calculations.

We also defined two metrics for measuring reasoning time:

- **Translation time:** This is the time taken for translating the IVML object model to the equivalent representation in the reasoner specific language. Currently, the reasoner integration produces an intermediary representation (a temporary file) and passes it to the reasoner for parsing it into the reasoner-specific representation. However, future versions may avoid temporary files e.g. using the Jess API or Drools knowledge providers.
- **Reasoning time:** For Jess, this is the time taken by the reasoner to parse the intermediary representation and to calculate the reasoning result. For Drools, we also explicitly show the **parsing time**, i.e., the part of the reasoning time, which is consumed by parsing and creating the internal knowledge base. However, Jess does parsing and reasoning in one step. Thus, we can only provide the combined reasoning time.

We collected these measures by executing each generated IVML ten times for both reasoners. For each execution both reasoners calculated the same results. The tests were executed on a Macbook Pro running Mountain OS Mountain Lion. The specifications of the machine are as follow: 2.9 GHz dual-core Intel Core i7, 8GB 1600MHz Memory. Table 3 summarizes the results in terms of the minimum (Min.), maximum (Max.), and average (Avg.) execution times (given in seconds).

In summary, both reasoners performed well, while Jess performs on average better than Drools when it comes to pure reasoning time irrespective of the complexity of the model and the number of variables. For example, Jess takes an average time of 0.44 seconds for the complex models (1000 variables and 100 constraints), while Drools takes 7.23 seconds for the same models. According to additional experiments with Drools we found that 34% of that time is consumed by the creation of temporary Java classes which represent the data in the working memory, 65% by parsing the rules while just 0.06 seconds are consumed by actual rule processing. Thus, Drools consumes nearly 99% of the reasoning time in the creation of the knowledge base with facts in memory (parsing time in Table 3). However, we believe that this performance issue of Drools can be resolved in the future, e.g., by directly passing the values from the IVML model into the reasoning process, e.g., by using an own knowledge provider.

3.4 Instantiation support for IVML

An instantiator is an external and maybe third-party tool that resolves variability in product line artefacts in its specific way. For example, the Velocity instantiator, which is provided as a basic instantiator of EASy-Producer, resolves Velocity-specific statements within Java code according to a specific configuration. This resolution capability allows deriving individual domain-specific platforms based on the configuration values. However, as IVML provides highly complex variability modelling capabilities, the instantiation process has to be adapted in order to provide support for all IVML elements relevant to the instantiation process. This includes binding times, production processes, etc., as described in Deliverable D2.2.1. In this section, we provide an overview on the current state of the instantiation support.

EASy-Producer provides two instantiators by default: the Velocity instantiator and the XVCL instantiator. Velocity is based on the Velocity Template Language (VTL)[29], which uses references in terms of VTL variables to manipulate Java code. Thus, in EASy-Producer the Velocity instantiator is implemented as a mapping and processing unit that first maps IVML variable values to VTL variables by their names and then processes the VTL statements according to these values. The XML-based Variant Configuration Language (XVCL)[30] instantiator is implemented similarly. However, this instantiator maps values of IVML variables to XVCL variables and statements in XML documents and processes them accordingly. Thus, we implemented the mapping strategies between configurable IVML elements and the individual instantiator languages for both instantiators.

Although SAP will also rely on the Velocity instantiator for generic deployment descriptors, the scenarios in the INDENICA-project require additional instantiators that process the generic artefacts in a technology-specific way. In the SAP yard management, the Cocktail approach and more specifically the Lombok[31] framework is responsible for runtime configuration. Thus, we implemented a specific EASy-instantiator, which supports the SAP build process. This instantiator maps IVML decision values and binding times to Cocktail annotations. Then the instantiator performs modifications of the compiled code (byte code) in order to enable compile time and runtime variability, i.e., source files are not modified.

Three different types of artefacts will be instantiated in the ***SIE warehouse case***,

- A Spring[32] service configuration. The Spring configuration indicates the actual services, service compositions and service specific parameters. Individual parameter values and active services will be derived from the variability model / configuration and turned into a Spring configuration by a specific instantiator.

[29] http://velocity.apache.org/engine/devel/user-guide.html#Velocity_Template_Language_VTL:_An_Introduction

[30] http://xvcl.comp.nus.edu.sg/cms/

[31] http://projectlombok.org/

[32] http://www.springframework.net

- The scripts for creating the warehouse database, which reflects the warehouse topology as it is configured in the variability model. A specific instantiator for this type of artefact will be realized.
- The configuration of the object-relational database mapper, which turns entries in the database tables to instantiated objects and in turn, causes appropriate adjustments of the user interface. The initial version of this instantiator will derive the artefact from a generic (template) artefact while an improved future version may be based on variation point specifications and XML transformations.

The instantiator plug-ins can be developed independently from the core of EASy-Producer. We provide a step-by-step developers guide[33] that describes the required implementation steps as well as the required configurations of Eclipse plug-in projects in order to guarantee a successful integration with EASy-Producer.

The instantiation components will be subject to an extensive regression test suite in the future. However, the implemented instantiators as described above, the developers guide as well as the integration mechanism are currently used successfully by the project partners.

Further, we will develop a new approach to the instantiation support as part of our work in INDENICA as initially described in Deliverable D2.2.1. Similar to IVML, we will design this approach in terms of a textual modelling language for the production process, namely the Variability Implementation Language (VIL). VIL will provide a simple, but expressive language for defining the build process of variable product line artefacts regardless of the variability implementation technique in use. This covers aspects like the definition and combination of build tasks as well as the integration with IVML. The resulting VIL-specifications will be processed by a VIL-engine, which resolves the variabilities according to the IVML configuration, such as the decision variable values, the binding time, etc. We developed first concepts that enable the definition of such VIL specifications. These concepts are currently tested against real-world build-processes used by the project partners, e.g., exchanging services at compile time and at runtime. However, these concepts and, thus, the elements of VIL are still under development.

3.5 Support for Application Engineers

The elements and concepts of the IVML language provide Platform Providers with high expressiveness and freedom in modelling the variability of a variant-enabled base platform. However, this needs deep domain and modelling knowledge as modelling a product line (of service platforms) may yield rather complex models. In contrast, Platform Variant Creators (in general Application Engineers) need less capability while configuring a valid domain-specific platform. Thus, we provide an explicit configuration editor as part of EASy-Producer in order to ease the configuration task. In this section, we will discuss our approach to support the Platform Variant Creator. Further, we will provide an overview on the current state of the implementation and validation.

[33] http://projects.sse.uni-hildesheim.de/easy/docs/devguide.pdf

The configuration editor in EASy-Producer is intended to be used by Platform Variant Creators for simple product configuration task. While the IVML editor can also be used for configuration, the configuration editor hides the underlying complexity of IVML models. This is achieved by the following design decisions:

- The configuration editor displays all available (and visible[34]) decision variables of the IVML model. This is different from the IVML-Editor, which does not display any model element by default. Thus, using the IVML-Editor for configuration requires domain knowledge about the available decision variables and their individual configuration space.
- The configuration editor displays the values that can be assigned to a decision variable of type Boolean, enumeration, and references. However, as the number of possible values for decision variables of type Integer and Real are in general infinite, the editor cannot provide complete configuration support for such variables. Thus, configuration support for Integer and Real variables is currently limited to displaying the decision variables. Further, compounds and containers will be displayed in terms of collapsible hierarchies to ease their configuration and reduce complexity
- The configuration editor displays nested elements, for example compound variables, as hierarchical structures. This improves clarity and understanding for related model elements as well as the impact of their configuration.

We also encountered some limitations with respect to providing good configuration support. In case of decision variables of type Integer and Real, displaying all valid values for assigning such variables is not possible due to the infinite configuration space mentioned above. However, in case of restricting the number of possible values by constraints (cf. value propagation in IVML reasoning), at least for Integer variables such a support will be available in the future. Further, displaying nested elements, such as compounds, in a hierarchical structure is limited in terms of a limited number of hierarchy levels that can be displayed in an understandable way. For example, if a nested compound element is in turn of type compound, etc., in some situations the number of hierarchy levels will exceed the boundaries of the configuration editor. Thus, configuring such complex hierarchies is difficult from a user's perspective. However, we are currently working on a suitable solution for these problems.

The configuration editor is currently available as an initial prototype within EASy-Producer. It is capable of editing decision values of the basic IVML types, such as Boolean, Integer, Real, and String. Further, decision values of complex types like compounds and containers can be edited.

[34] In case of using configuration interfaces in IVML, only the decision variables of imported project interfaces are visible and available for configuration (cf. Deliverable D2.1).

4 Integration with other WPs

In this section, we will discuss how the work on the Variability Engineering Tool integrates with the work done in the other WPs. In WP2, integration mainly happens through defined software interfaces, instantiation of generic artefacts and reuse of components. In this section, we will detail the already realized integrations as well as future integration tasks. All work described in this section was originally done within the INDENICA project.

For the integration with **WP1**, we offer specific integration components for IRET. These components enable IRET to create a variability model based on the variability-related information contained in IRENE goal models. However, as IRENE is a requirements engineering methodology rather than a variability modelling approach, the information captured in IRENE only leads to an initial variability model to be completed later by the Platform Provider. Based on the variability information captured in IRENE, the integration components for IRET provide functionality to:

- Create an IVML variability model.
- Add a decision variable to an IVML variability model. A decision variable is specified as strings stating the name of the variable, the type and an optional default value. Unknown types are converted to empty IVML compounds.
- Define constraints on the created decision variables using the IVML constraint syntax. The syntactic and semantic validity of the constraints is ensured with respect to the already defined decision variables and types.

In contrast to IRET models, which may serve as an input to variability modelling, the selections to be made in the architecture decision support framework ADF may be subject to variability. For example, the configuration of a high-performance service based system may disable several design patterns in the architectural decision support tool. Thus, the integration needs to provide:

- Read-only access to information in an existing variability model.
- Evaluation of presence conditions. A presence condition is an IVML (constraint) expression which indicates whether optional or alternative parts (here design decisions) shall be enabled or disabled as specified in a given variability configuration.

Similar to the architecture decision support framework ADF, also certain parts of the architecture created with the view-based modelling framework VMBF developed in **WP3** may be subject to variability. For example, as a consequence of disabling several alternatives in the architectural decision framework, related elements shall not be part of the architecture in VBMF. As both the architecture decision framework and VMBF require similar integration functionality, we provide this in terms of the UniVie integration, i.e., two components, one containing the interface and one the EASy-based implementation.

Two specific types of integration are needed for the INDENICA runtime environment developed in **WP4**, namely artefact-based and connector-based integration:

- Several customization aspects of the runtime environment for a virtual service platform will be described in terms of configuration artefacts, such as monitoring and adaptation rules. Some configuration artefacts will be generic in order to be reusable or adaptable, i.e., individual parts may be enabled or disabled according to a given (runtime) variability configuration. The variable parts in the generic artefacts will be annotated by selector expressions and presence conditions and specific instantiation mechanisms for monitoring and adaptation rules will be provided. Further, we developed a specific tool integration component for the WP4 rule editors so that the Platform Architect can be supported via content assist mechanisms showing available IVML decision variables.
- The runtime environment will contain the following (headless) components of the EASy architecture: IVML core, IVML model, instantiator core, the specific instantiator (for rule instantiation at runtime), the reasoner core and a specific reasoner integration. Due to the flexibility of the reasoner core, i.e., the ability of handling multiple reasoners, the INDENICA runtime environment does not need to rely on a fixed reasoner. Here, we aim at providing an optimized reasoner integration based on analyzing the open variabilities in the partially instantiated variability model of the virtual service platform.

Regarding the base platforms developed in **WP5**, specific instantiation mechanisms have or will be realized. This includes

- An instantiator for the ***SAP yard management case***. At runtime, the generic SAP online configuration mechanism enables administrative users to change open variabilities dynamically. This generic mechanism is also generated from the information provided by the Cocktail instantiator to Cocktail / Lombok as part of the SAP build process. However, the user input may provide an invalid input to this configuration mechanism, i.e., a configuration which conflicts with the constraints in the variability model. We support the necessary validation in terms of a specific integration component which is structurally similar to the WP1/WP3 integration but realizes a runtime connector to the SAP cloud system and, including the related EASy components, fulfils the specific requirements of the SAP server runtime environment.
- Three different types of artefacts will be instantiated in the ***SIE warehouse case*** as already described in Section 3.4.

5 Conclusion

In this deliverable, we characterized the state of realizing the INDENICA variability engineering tool EASy-producer for customizing service platforms. We discussed the practical application of the tool in terms of an introductory example, the architecture and its logical components as well as their individual realization state and, finally, the integration with the other WPs.

For most of the logical components of the INDENICA variability engineering tool we already provide almost fully functional realizations. This is in particular the case for the core components such as the IVML core, the IVML object model, and the reasoning core as well as the integration components. Further, we implemented various instantiation mechanisms as well as two distinct reasoner integrations. Both reasoner integrations cover approximately 80% of the concepts defined in the INDENICA Variability Modelling Language at already good performance.

In accordance with the work plan of the project, there is still significant work to be done to complete the INDENICA variability modelling tool. As discussed in this deliverable, this work consists of tasks such as completing and optimizing the reasoning for IVML, refining the instantiation capabilities for the individual partners and realizing those for the TEL remote maintenance case, providing headless functionality for seamless integration with the WP4 runtime environment as well as realizing improved support for the Platform Provider and Platform Variant Creator in terms of simplifying modelling editors. Further, the instantiation core will be refined and extended based on the concepts of the Variability Implementation Language (VIL) to be presented in D2.2.1. VIL specifications will in particular support the (headless) integration into the build and deployment processes used in service-based systems and service platforms.

In summary, the realization work done in WP2 is in line with the work plan of the INDENICA project and major steps for realizing the INDENICA variability engineering tool have been successfully carried out. Optimizations of the existing components as well as refining specific components such as the realization of VIL will be done in the third year of the project.

References

[1] L. Chittaro, R. Ranon, E. Carchietti, A. Zampa, E. Biasutti, L. Marco, and A. Senerchia. A Knowledge-Based System to Support Emergency Medical Services for Disabled Patients. In *Proceedings of the 12th Conference on Artificial Intelligence in Medicine (AIME '09)*, pages 176–180, 2009.

[2] K. K. L. Ho and M. Lu. Web-based expert system for class schedule planning using JESS. In *Proceedings of the 2005 IEEE International Conference on Information Reuse and Integration (IRI '05)*, pages 166–171, 2005.

[3] I. Horrocks, P. F. Patel-Schneider, and F. van Harmelen. From SHIQ and RDF to OWL: the making of a Web Ontology Language. *Journal of Web Semantics*, 1(1):7–26, 2003.

[4] M. Mendonça, A. Wasowski, and K. Czarnecki. SAT-based Analysis of Feature Models is Easy. In *Proceedings of the 13th International Conference on Software Product Lines (SPLC 2009)*, pages 231–240, 2009.

[5] B. Mu, F. Xiao, and S. Yuan. A rule-based disease self-inspection and hospital registration recommendation system. In *Proceedings of the 3rd International Conference on Software Engineering and Services Science (ICSESS '12)*, pages 212–215, 2012.

[6] M. O'Conner, H. Knublauch, T. Samson, and M. Musen. *Writing Rules for the Semantic Web Using SWRL and Jess*, 2005. Workshop Protégé with rules, Madrid, Spain.

[7] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Variability Model of the Linux Kernel. In *4th International Workshop on Variability Modeling of Software-intensive Systems (VaMoS 2010)*, 2010.