



Engineering Virtual Domain-Specific Service Platforms

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

Variability Engineering Tool (final)

Abstract

Domain-specific customization of service platforms can only be effective with appropriate tool support. In this deliverable we will present the final state of the INDENICA Variability Engineering tool. The focus of this deliverable is on the evolution of the tool since the intermediate Deliverable D2.4.1. This includes a discussion of the general enhancements to the tool, and, in particular, the Variability Implementation Language (VIL) as a major extension for instantiation support. Based on these enhancements, we will revisit the running example introduced in Deliverable D2.4.1 to illustrate the changes made to the tool. A major part of this deliverable is also contained in the form of appendices, which include the documentation of the Variability Engineering Tool.

Document ID:	INDENICA – D2.4.2
Deliverable Number:	D2.4.2
Work Package:	WP2
Type:	Deliverable
Dissemination Level:	PU
Status:	final
Version:	1.0
Date:	2013-09-30
Author(s):	SUH, SAP

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	30. Sep. 2012	final revision and corrections of complete document
1.0	30. Sep. 2012	final version

Document Properties

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

Table of Contents

1	Introduction	5
2	Variability Engineering Tool Overview	7
2.1	Evolution of EASy-Producer beyond D2.4.1	7
2.2	Textual Instantiation Support	8
3	Variability Implementation Language.....	11
4	Running Example – Revised	14
4.1	Configuration Space Definition Including Comments	15
4.2	Implementation Space Definition Using VIL	16
4.3	Improved Configuration of a Domain-Specific Service Platform	21
5	Conclusion.....	22
6	References	23
7	Appendix: EASy-Producer Documentation	24
	Appendix 1: EASy-Producer User Guide	25
	Appendix 2: EASy-Producer Developers Guide	51
	Appendix 3: IVML Language Specification	84
	Appendix 4: VIL Language Specification	153

Table of Figures

Figure 1: Overview of the VIL instantiation process.....	9
Figure 2: Comments for configuration support in EASy-Producer.....	16
Figure 3: VIL-build script in EASy-Producer.	17
Figure 4: VIL-build script of the content-sharing application.	18
Figure 5: Cocktail variability model template in VIL.	20
Figure 6: Cocktail resolution model template in VIL.....	20
Figure 7: Product configuration using the IVML configuration editor.	21

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, in particular in deliverables D2.1, D2.2.1 and D2.2.2. This deliverable supplements the interim deliverable D2.4.1 and addresses in particular the introduction of the VIL-language, which was not represented in D2.4.1. However, we refrain from reproducing the main content of D2.4.1 here for the sake of brevity.

In Section 2 we will discuss the main evolution steps that have been performed with respect to the tool environment in comparison to deliverable D2.4.1. The focus is, however, on changes that were made, the basic state as described in deliverable D2.4.1 will not be reiterated. This section also outlines some of the basic ideas and decisions relevant to the VIL. Next, Section 3 discusses the main components of the VIL. Finally, Section 4 provides a running example, which is used to illustrate the current and final state of the variability implementation tool. We reuse the same example as has been used in Deliverable D2.4.1 as this helps to highlight the differences in the revised version.

A major part of this deliverable is also contained in the form of appendices. There, we provide a number of guidebooks that have been developed to supplement the variability tool implementation and support users (including developers as users) in employing the technology provided in the form of this tool. We attach four supplementary documentations: the Users Guide and the Developer Guide for the Easy-Producer Tool and a language specification for the IVML and VIL language respectively.

Further relationships to other INDENICA deliverables are:

- Deliverable 1.3.1 and 1.3.2 discuss the realization of the INDENICA decision framework. In particular, deliverable 1.3.2 discusses the integration of variability decisions and architectural decisions and the role of the Variability Engineering tool EASy-Producer as part of it.
- Deliverable 2.1 discussed the basic requirements of the variability modelling approach and the basic concepts that were introduced within the INDENICA Variability Modeling Language (IVML). The implementation of this has already been introduced with Deliverable D2.4.1.
- Deliverable 2.2.1 and 2.2.2 discussed the conceptual basis that lead to the variability implementation language VIL, which has been now realized and is introduced as part of this deliverable.
- Deliverable D5.4 discusses the evaluation and assessment of the INDENICA technologies, including the EASy-Producer tool.

Comments on the relation to previous work:

- The variability implementation language has been exclusively created as part of the INDENICA development, with a strong focus on addressing INDENICA requirements as described in previous deliverables. As there is very little in this direction from other research, we regard this as a highly novel contribution by this project.
- The variability engineering tool profited to some extent from previous work (partially predating INDENICA), but most of the implementation had to be rewritten during the project to accommodate INDENICA requirements.

2 Variability Engineering Tool Overview

In this section, we will discuss the final state of the variability engineering tool EASy-Producer with a particular focus on the enhancements relative to the state described in Deliverable 2.4.1. This includes in particular the enhancements that have been made with respect to the INDENICA Variability Modelling Language (IVML) and the introduction of the Variability Implementation Language (VIL).

We will structure this discussion as follows: Section 2.1 will provide a brief overview of major categories of general changes and improvements in EASy-Producer. In Section 2.2, we will discuss the introduction of the VIL and its integration into the EASy-Producer tool platform. A detailed discussion on the design and the implementation of VIL will be provided in Section 3.

2.1 *Evolution of EASy-Producer beyond D2.4.1*

While the largest area of change and enhancement is the introduction of the VIL, also several enhancements have been made in other areas. Below we discuss the major areas in which improvements were made:

- **INDENICA Variability Modelling Language (IVML):** IVML was first introduced in Deliverable D2.1 and fully defined in Deliverable D2.4.1. Thus, only minor changes and improvements were applied to the language concepts, like the clarification of certain modelling elements. Further, some elements that increase the usability of the language were added, like the mass-assignment of attribute values. The IVML language specification in the appendix provides a full description of the modelling elements available in the final tool.
- **Reasoning support for IVML:** The reasoning support for IVML was a major milestone in Deliverable D2.4.1. We implemented two alternative rule engines to support the configuration of valid product instances based on a variability model defined in IVML. While the first implementations only checked for validity of certain configurations, the Drools reasoner also supports value propagation to actively support the configuration process. Value propagation enables the assignment of decision variables with valid values by the reasoner based on the values previously assigned to other decision variables. The reasoner calculates these values based on the constraints defined in the variability model and assigns values if they are uniquely implied by other constraints. This eases the configuration task as the user needs to provide fewer configuration items. In particular, a complete and valid configuration can be automatically derived based on a subset of manually assigned variables (if they are related by constraints).
- **Configuration support:** The IVML language provides highly expressive modelling elements and concepts for the definition of variability models. Thus, defining a valid configuration based on a variability model is a rather complex task. In order to support application engineers, we improved and extended the editors of EASy-Producer to ease the configuration of a specific product and to provide guidance throughout the product derivation process.

The mayor improvements regarding the configuration support are the value propagation introduced above, the ability to provide additional comments to explain the purpose of decision variables and configuration options, editor-based addition and deletion of new elements to container variables, and feedback regarding the individual steps as well as the success or failure of the instantiation process. We will also illustrate these improvements in the running example in Section 4.

The main focus regarding the general improvements of EASy-Producer is on the usability of the tool and the guidance to users. Further improvements affect the reasoning and (to some extend) the IVML language as described above. However, the major evolution of EASy-Producer is in the instantiation support as we will discuss below.

2.2 *Textual Instantiation Support*

The instantiation support of EASy-Producer in its earlier versions was limited to the application of instantiators to a (subset) of individual artefacts of a software product line (cf. Deliverable D2.4.1). Also, in order to create a new instantiator, it had to be implemented specifically as Eclipse plug-in, and explicitly integrated into the EASy-Producer tool. Due to the need of a more flexible definition of the complete instantiation processes and the absence of adequate third-party tools or concepts, we introduced the concept of the Variability Implementation Language (VIL) in Deliverable D2.2.2. In this section, we will provide an overview of the integration of VIL into EASy-Producer. A detailed discussion regarding the design and the basic concepts of VIL will be provided in Section 3.

VIL provides a simple, but expressive language for defining the instantiation 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 are processed by a VIL-engine, which resolves the variabilities according to the IVML configuration, such as the decision variable values, the binding time, etc., as illustrated in Figure 1. Further, the instantiators of EASy-Producer as well as any third-party tools can be integrated into VIL in order to reuse existing variability realization techniques, like the instantiators described in Deliverable D2.4.1.

VIL significantly goes beyond the instantiation support that existed in EASy-Producer (and other variability management tools) so far. Below, we will briefly describe the main advantages of VIL with respect to the previous instantiation support of the tool:

- **Artefact management:** The management of generic artefacts as well as specific (instantiated) artefacts in EASy-Producer was mostly implemented in the instantiator core component (cf. Deliverable D2.4.1). Further, EASy-Producer was initially designed for Java-projects, which required changes to the implementation of the tool or at least the instantiators to support other types of projects and artefacts. Also selecting the application of instantiators was restricted to a “per-artefact” base and the instantiators were then applied to each artefact individually (taking the configuration into account). These artefacts (including directories) had to be visible in the project at

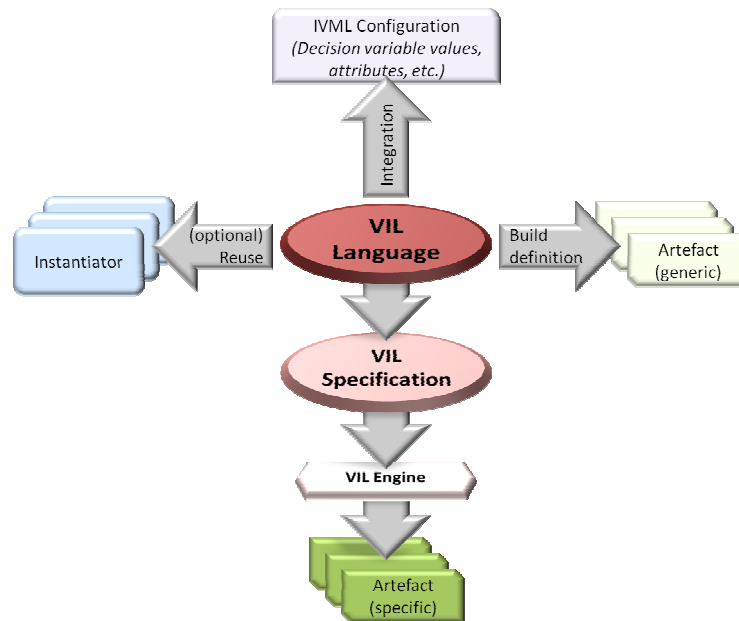


Figure 1: Overview of the VIL instantiation process.

definition time. The handling of the artefacts, e.g. creating, deleting, or copying artefacts, was implemented in the instantiator core component and could not be changed by the domain engineer. Finally, if more than one elementary instantiator should be used for an artefact the order of application could not be explicitly controlled. This has been changed by the introduction of VIL in several ways:

- **Artefact selection in VIL:** VIL enables domain engineers to freely select artefacts of any type to be processed in the instantiation process. These artefacts can be placed at any location and must not explicitly be part of the actual EASy-Producer project. This definition is also evaluated during runtime of the instantiator, which allows to first create artefacts, which are then instantiated in subsequent steps.
- **Artefact handling in VIL:** Domain engineers decide how to handle specific artefacts. For example, domain engineers specify when and how to take over the generic artefacts of a software product line into a specific project and when to apply which production strategy. This also allows handling artefact inheritance in more flexible ways than before.¹
- **Artefact manipulation:** The manipulation of artefacts was defined in the individual instantiators of EASy-Producer. For example, the creation of new artefacts was part of the instantiator implementation. Thus, defining new manipulations or changing existing manipulations required the modification of the instantiator implementation. This has been simplified greatly by the introduction of the VIL.
 - **General artefact manipulation in VIL:** The creation of new artefacts and the deletion of existing artefacts can be defined as part of a VIL

¹ The previous mechanism only allowed to either state that all artefacts of a certain project need to be copied or only a reference had to be made. No mix was allowed.

specification. This enables the domain engineer, for example, to generate new artefacts as part of the instantiation process with only one line of code in the VIL specification.

- **Artefact content manipulation in VIL:** The manipulation of artefact contents can be defined in terms of so-called VIL templates. These templates define a generic schema, for example, for changing the content of a specific type of artefact in accordance to a specific product configuration. These templates can be defined using the VIL template language (see Section 3). However, one is not restricted to this, but rather also external programs can be used to perform content manipulation.

We will illustrate these advantages in the running example in Section 4.

3 Variability Implementation Language

In this section, we will discuss the design and the basic concepts of the INDENICA Variability Implementation Language (VIL). The concepts of VIL were initially introduced in Deliverable D2.2.2. Due to practical experience in actually applying these concepts to the different INDENICA case studies, we revised some of these concepts and developed further concepts that enable a more comprehensive specification of the required instantiation processes. These concepts are designed to realize instantiation of artefacts in a generic, but simple way. Further, VIL still consists of the four main constituents, namely the artefact model, black-box instantiators, the VIL build language, and the VIL template language, which we will discuss in detail here.

The decision to create two different languages in VIL was introduced to create a separation of concerns. One language for the customization and generation of individual artefacts (VTL), while the other focuses on the overall orchestration of the creation process. In particular VIL aims at describing elementary production strategies as introduced in Deliverable D2.2.1 and combining them into more complex derived strategies. A production strategy, as we defined it in Deliverable D2.2.1, defines the instantiation of a specific type of artefacts at a specific binding time. One approach to this is to use the *VIL Template Language* (in particular for textual artefacts), which enables the definition of artefact generation as well as artefact transformation as part of a production strategy in a reusable way. However, the definition of a production strategy also requires the relation of the artefacts and the instantiation mechanisms. Further, a software product line (or a generic service platform) typically consists of multiple different types of variable artefacts and, thus, requires the combination of production strategies at the same binding time or even at different points in time. For this purpose, we designed the *VIL Build Language*. This language provides essential modelling elements for the specification of production strategies, their combination, as well as further tasks that must be executed as part of a complete instantiation process.

The *VIL Template Language* combines capabilities of popular generator or template languages such as Xtend [2], Xpand [1] and Apache Velocity [5]. Although the VIL template language is rather closely related 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 [3] language infrastructure generation).

The *VIL Build Language* substitutes the VIL workflow language introduced in Deliverable D2.2.2 and enables the specification of instantiation processes as well as individual production strategies in terms of rules (similar to make [4]). The decision towards a rule-based approach relies on the following advantages of rules over workflows:

- **Pre- and post-conditions:** A rule may explicitly describe its pre- and post-conditions (these conditions are optional in VIL), which must be fulfilled

before the execution of the rule, or after the execution respectively. In the workflow-based approach intended earlier such conditions either required explicit definitions as part of the actual workflow implementation, or were not possible at all (i.e., definition of post-conditions as part of the respective workflow).

- **Reduced coupling:** The pre- and post-conditions of a rule further support implicit rule calls as part of an execution. For example, a rule r_1 may define the presence of an artefact A as a pre-condition, while another rule r_2 defines A as its post-condition. In this scenario, the VIL engine will automatically search for a rule that satisfies the pre-condition of r_1 before executing this rule and, thus, will execute r_2 first. In the workflow-based approach, such implicit relations could not be defined at all.
- **Implicit iterations:** The VIL build language supports the definition of generic patterns instead of explicit artefacts. For example, the pattern `src/*.java` may be used to identify all Java-file artefacts in the `src`-folder of a certain implementation. Further, this pattern can be used to define the application of operations on each of the Java-files identified by this pattern. In the workflow-language, we had to define explicit loops (that may include additional conditions) to identify a specific subset of artefacts.
- **Partial execution:** A typical capability of rule-based system like make [4] is the ability to automatically calculate which artefacts are up-to-date and which artefacts must be updated. This enables the partial execution of those rules of a build-script where the target artefacts are not up-to-date (instead of executing the complete build-script). A workflow-based approach cannot provide such a capability at all.

The VIL build language provides very expressive modelling concepts, for example, it allows to combine parameterization with rules, allows artefact matching, has functional and declarative language elements, etc. (see the VIL language specification in Section 7). A particular capability that is unique is the integration of IVML concepts, like decision variables, attributes, etc., which define the scope of an instantiation and configure the transformation or generation of certain artefacts.

The actual manipulation of artefacts (as part of the template language as well as of the build language) requires the availability of certain operations on artefacts. However, the operations available heavily depend on the type of artefact, which makes a generic implementation of operations for all artefacts impossible. Thus, we developed an explicit *Artefact (Meta-) Model* as the foundation of VIL. As introduced in Deliverable D2.4.1, this model provides all (currently²) supported types of artefacts and their operations that can be used in the VIL languages (and in the blackbox instantiators, which we will describe below).

The last conceptual decision regards the integration of VIL with other tools, like available compilers or linkers, or already implemented instantiators (like the ones described in Deliverable D2.2.2). The resulting concept of *Blackbox Instantiators* in

² The VIL artefact model can be extended by new artefact types and their operations. A description of how to extend the artefact model is provided by the EASy-Producer Developers Guide in Section 7.

VIL allows the integration of such tools in two ways: a) the execution of existing (third-party) tools may be explicitly defined as part of a VIL build script (similar to command line executions), or b) such tools may be wrapped into a VIL extension³. This concept enables the application of VIL to legacy software product lines or other software projects, which already use certain kinds of instantiation mechanisms.

³ The implementation of new instantiators is described in the EASy-Producer Developers Guide in Section 7.

4 Running Example – Revised

In this section, we will revisit the running example introduced in Deliverable D2.4.1 and use it to describe the changes that have been made to EASy-Producer discussed in Section 2. Thus, we will not describe the complete example in all details. Rather, we will start with a summary of the individual steps discussed in Deliverable D2.4.1 and will then discuss the changes in the steps that are affected by the tool evolution.

In the running example given in D2.4.1, we illustrated and discussed the definition of a software product line from which multiple variants of a content-sharing platform can be derived. This included the following steps:

1. **Definition of a new base service platform:** 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. This includes the definition of a new product line project, the configuration space, and the implementation space.
 - 1.1. **Configuration space definition:** The definition of a variability model using IVML, which defines the configuration space of a specific software product line. While the basic definition of a variability model does not change, we will illustrate the definition of additional comments for decision variables to support the configuration task of application engineers (see Step 2.1) in Section 4.1.
 - 1.2. **Implementation space definition:** This includes the implementation of the generic artefacts, including a specific variability implementation technique and the application of a specific instantiator for the instantiation process. While the implementation of the generic artefacts is the same, the application of an instantiator will be substituted by the definition of a VIL build script and two VIL templates in Section 4.2.
2. **Derivation of a domain-specific service platform:** The process of deriving a new domain-specific service platform from a software product line defined in EASy-Producer from the perspective of a Platform Variant Creator. This includes the configuration and instantiation of a domain-specific service platform.
 - Configuration of a domain-specific service platform:** The configuration of a specific service platform in terms of assigning values to the decision variables of the variability model defined in Step 1.1. In general this step does not change. However, we will illustrate the increase in usability of the tool and the support of the application engineer in Section 4.3.
 - 2.2. **Instantiation of a domain-specific service platform:** The final instantiation of the domain-specific service platform based on the configuration defined in Step 2.1. This step does not change.

We will use the following font styles throughout the following sections 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`.

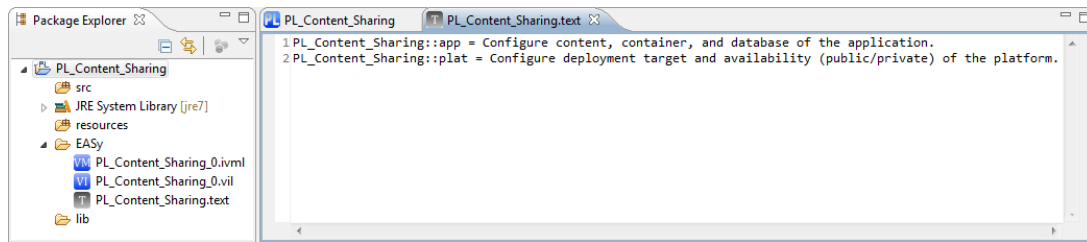
4.1 Configuration Space Definition Including Comments

The definition of the configuration space in terms of a variability model using IVML is typically done by domain experts. These experts know what the different configuration options mean and how they must be configured to yield a valid product. However, this knowledge is typically not available for application engineers that perform the task of configuring the final products. Thus, EASy-Producer supports additional text-files that can be used to define comments on IVML decision variables, e.g., to describe the impact of a certain configuration. In this section, we will illustrate how to define such a text-file to guide the Platform Variant Creator through the configuration of a specific service platform in Section 4.3.

The first step is to create a new text-file in the **EASy-folder** of the product line project. For this purpose, right-click on the **EASy-folder** and select *New* → *Other...* → *EASy-Producer* → *IVML Comments File*. The name of the file must match the name of the IVML model file² in order to unambiguously link the comments to the desired decision variables. Thus, enter `PL_Content_Sharing_0.text` as the name of the new file and click the *Finish* button. We will open the new text-file with a simple text-editor to enter the comments. Each comment-definition starts with the name of the IVML project⁴ followed by “::”, the name of the decision variable for which we want to define the comment, and an equal-sign (“=”). The actual comment is defined in plain text after the “=”. It ends at the end of the line.

The result of the comment-definition is shown in Figure 2. We defined two comments for the two decision variables “app” and “plat” of the running example (cf. Figure 2a)), which are displayed in the **IVML Configuration Editor** of the **Product Line Editor** (cf. Figure 2b)).

⁴ Please note that a product line project may include multiple IVML-files, e.g. in a multi software product line scenario. Further, an IVML-file may include multiple project-definitions. Thus, the name of the project must be part of the comment-definition for a specific decision variable. However, in our example the names of the product line project, the IVML-file and the project-definition in the file are rather similar.



a) Comment-definition in the text-file.

Filtering Options					
Decision Name	Current value	+	-	Freeze	Comment
app	UNDEFINED				Configure content, container, and database of the application.
plat	UNDEFINED				Configure deployment target and availability (public/private) of the platform.

b) Comments in the IVML configuration editor

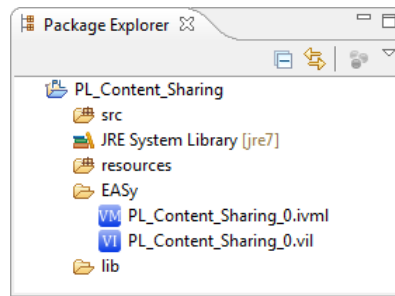
Figure 2: Comments for configuration support in EASy-Producer.

While the definition of comments for decision variables seems to be no big deal, comments support the configuration of valid products in an easy, but effective manner. Further, these comments provide a certain kind of documentation of the configuration options of software product line, which can be maintained over the complete lifecycle of the product line.

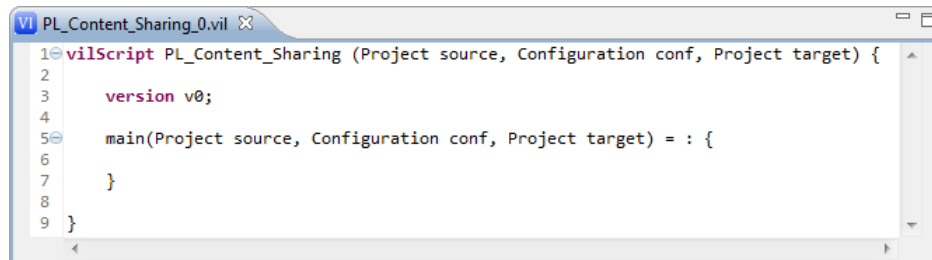
4.2 Implementation Space Definition Using VIL

In the running example we illustrated the implementation space definition using SAP's Cocktail instantiator, which was specifically developed for the SAP yard management use case. We demonstrated Cocktail for base platforms developed in Java and expressed 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. Based on this implementation, we will describe the definition of a VIL-build script and according VIL templates in this section.

VIL consists of two sub-languages, namely the VIL build language and the VIL template language. The VIL build script (defined with the VIL build language) is automatically created when a new product line project is created. In Figure 3a) the VIL-build script is named **PL_Content_Sharing_0.vil** and the initial content contains the mandatory top-level `vilScript` and an empty `main-rule` (cf. Figure 3b)). We will discuss these elements in detail below. Further VIL templates have to be created manually as they are only required if new artefacts must be created or existing artefacts must be modified in accordance to a specific schema (cf. Section 3). Thus, we will start with the definition of the build script in the existing VIL-file in the **EASy-folder** and then create the required templates.



a) The VIL-build script file.



b) The initial contents of a VIL-build script file.

Figure 3: VIL-build script in EASy-Producer.

The first step to define the build script for the content-sharing platform is to open the VIL build language editor by simply double-clicking the VIL-file **PL_Content_Sharing_0.vil** in the **EASy**-folder. The name of the file is composed of the name of the product line and the version number (here initially “0”).

By default, each VIL build script has a mandatory `vilScript` element and a mandatory version as shown in Figure 3b). The `vilScript` element is the top-level element of each VIL build script. The mandatory parameters of this element are:

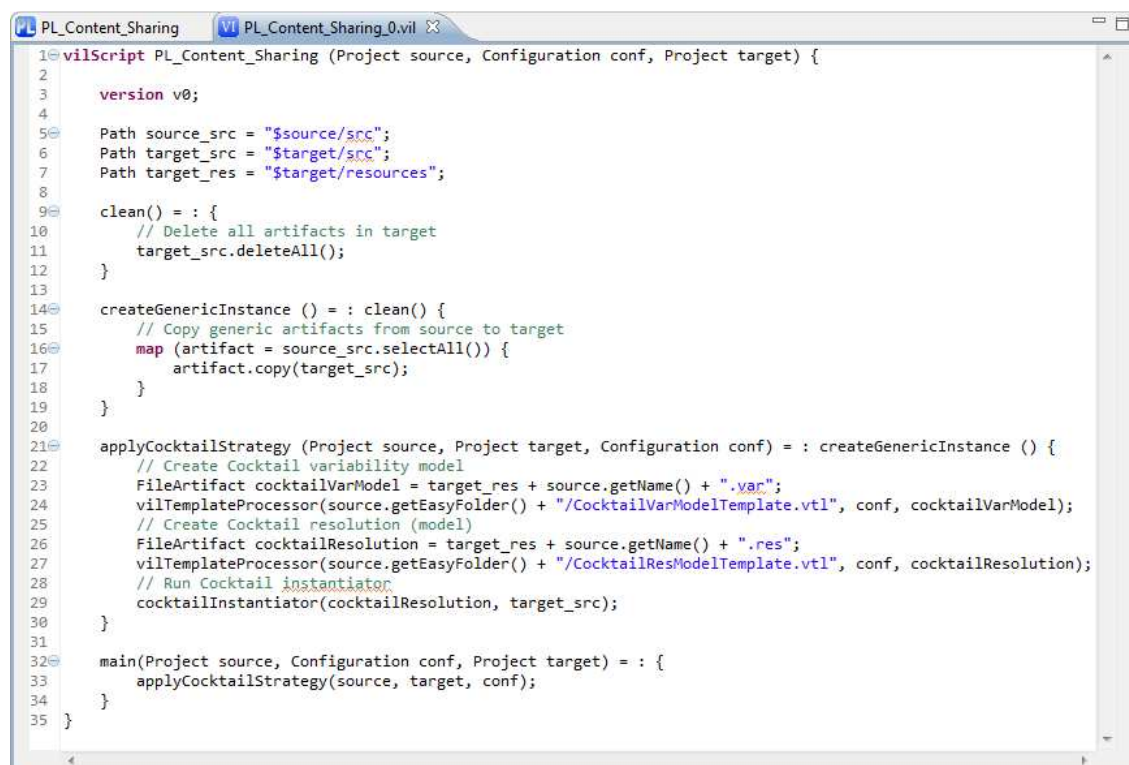
- `Project source`: the source project, which provides the generic implementation, for example, of a specific software product line. In this case the source-project is the generic content-sharing service platform.
- `Configuration conf`: the configuration of a specific product based on the variability model defined in IVML.
- `Project target`: the target project, which represents the specific product of a software product line, in this case of the content-sharing platform.

These parameters are automatically filled as part of the processing by the runtime environment. Further parameters can be added, but they then need to explicitly receive corresponding values.

Further, each build script contains a main rule (`main`), which is executed if no other rule is explicitly called by the instantiation process (which is the default in EASy-Producer). The main rule also accepts the source and the target project as well as the configuration as parameters. The equal-sign introduces the post-condition and the pre-condition separated by a colon. In the default build script in Figure 3b) no conditions are specified. This is the reason for “ : ” in Figure 3b). The actual content of the rule is then specified within the curly brackets.

Figure 4 shows the build script specifying the instantiation process of the variant-enabled content-sharing base platform. First, we define several paths to the source and target source-folder (`src`) as well as the resource-folder of the target project (lines 5-7). This eases the definition of the following rules as these paths are global and can be used by all rules in this script. For example, the rule `clean` (lines 9-12) uses the path to the target source-folder to delete all artefacts in this folder. The next rule (lines 14-19) triggers the `clean` rule as this is defined as a precondition of the rule. This means that we want to delete all artefacts in the target source-folder that were previously instantiated to ensure that only recent artefacts are available after the instantiation process. If this precondition is fulfilled (cleaning was successful), the rule is executed which yields the copying of the source artefacts to the target's source folder. The third rule (lines 21-30) defines the actual resolution of variation points in the generic artefacts (thus, rule `createGenericInstance` is a precondition to this rule). First, the Cocktail variability model and the Cocktail resolution model are created. This is done by creating a new `FileArtifact` in the target project for each of the models and using the VIL Template Processor together with respective templates to fill the new artefacts with their specific contents. Next, the Cocktail instantiator is executed which requires the Cocktail resolution model and the path to the (generic) target artefacts. Finally, in the main rule (lines 32-34) the `CocktailStrategy` is invoked.

Please note, that we could have replaced the main strategy directly with the code of the `CocktailStrategy`, making for a simpler implementation, but we wanted to illustrate that VIL-rules are more than classical rules (e.g., in `make`): they can either be invoked implicitly as preconditions – or explicitly like method calls.)



```

1 vilScript PL_Content_Sharing (Project source, Configuration conf, Project target) {
2
3     version v0;
4
5     Path source_src = "$source/src";
6     Path target_src = "$target/src";
7     Path target_res = "$target/resources";
8
9     clean() = : {
10         // Delete all artifacts in target
11         target_src.deleteAll();
12     }
13
14     createGenericInstance () = : clean() {
15         // Copy generic artifacts from source to target
16         map (artifact = source_src.selectAll()) {
17             artifact.copy(target_src);
18         }
19     }
20
21     applyCocktailStrategy (Project source, Project target, Configuration conf) = : createGenericInstance () {
22         // Create Cocktail variability model
23         FileArtifact cocktailVarModel = target_res + source.getName() + ".var";
24         vilTemplateProcessor(source.getEasyFolder() + "/CocktailVarModelTemplate.vtl", conf, cocktailVarModel);
25         // Create Cocktail resolution (model)
26         FileArtifact cocktailResolution = target_res + source.getName() + ".res";
27         vilTemplateProcessor(source.getEasyFolder() + "/CocktailResModelTemplate.vtl", conf, cocktailResolution);
28         // Run Cocktail instantiator
29         cocktailInstantiator(cocktailResolution, target_src);
30     }
31
32     main(Project source, Configuration conf, Project target) = : {
33         applyCocktailStrategy(source, target, conf);
34     }
35 }

```

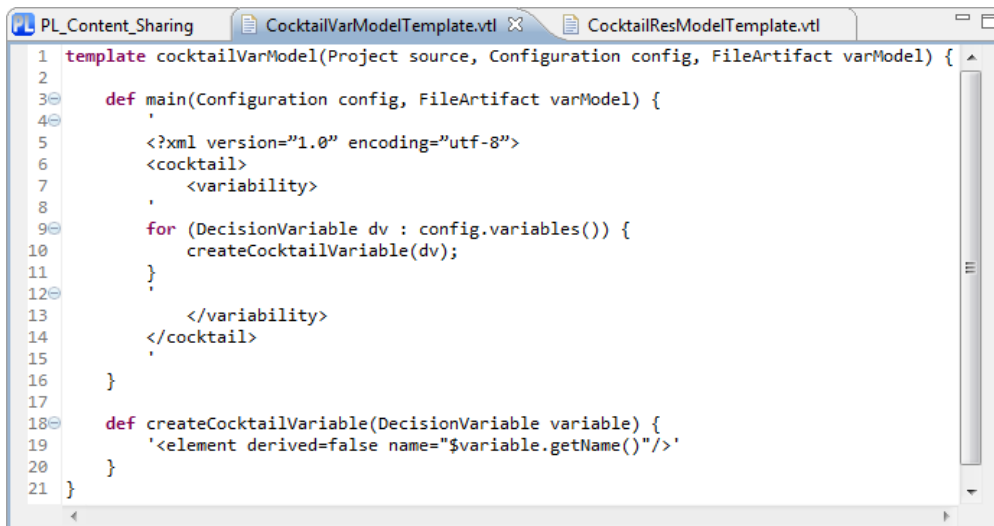
Figure 4: VIL-build script of the content-sharing application.

The next step is to define the two templates that are used in the build script of the content-sharing platform. Thus, we will first create the two files and then define their content. For this purpose, right-click on the **EASy-folder** and select *New* → *File*. We will use `CocktailVarModelTemplate.vtl` as the name of the new file (cf. Figure 4, line 24) and click the *Finish* button. We will do the same actions again but name the file `CocktailResModelTemplate.vtl` for the second template (cf. Figure 4, line 27).

EASy-Producer provides a specific VIL template language editor for the definition of VIL-templates that opens by double-clicking a template-file. Figure 5 shows the Cocktail variability model template in the VIL-template language editor. The `template-element` is the top-level element of each VIL-template. By default, this element receives the following parameters:

- `Project source`: the source project, similar to the VIL-build-script.
- `Configuration config`: the configuration of a specific product, similar to the VIL-build-script.
- `FileArtifact varModel`: in general, the last parameter is the artefact that will be created or manipulated by the template. In Figure 5 this is the file artefact, which will represent the Cocktail variability model.

The next mandatory element of a VIL-template is the main sub-template (lines 3-16). This element is the entry-point for the application of the template to an artefact by the VIL-template processor. In the example sub-template `main`, first, an XML-header as well as two opening XML-elements are defined (lines 5-7). As these XML-elements are surrounded by single quotation marks, they will be interpreted as plain text that is written to the content of the artefact (the same will be done in case of lines 13-14). Within the plain text definitions, a loop calls another sub-template named `createCocktailVariable` for each decision variable of the configuration (lines 9-11). This sub-template includes a single line of text indicated by the single quotation marks (line 19). However, the dollar-sign enables the use of a VIL function call, which will be interpreted and executed by the VIL template processor. This means, that the actual value of the attribute `name` will be the name of the decision variable passed as parameter to the sub-template. In this example, we assume that the names of the decision variables match the names of the resolution-elements expected by Cocktail. However, it would also be possible to use other names for the decision variables and to define a translation of the names to the names of the Cocktail resolution model as part of the VIL scripts. The final result will be the addition of an `element-element` for each decision variable of the configuration within the `variability-element`.



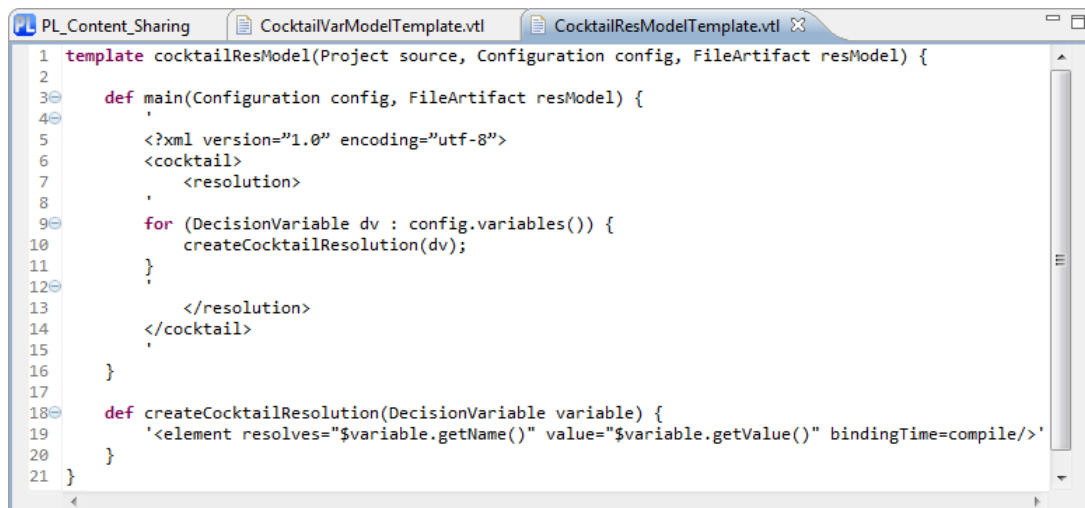
```

1  template cocktailVarModel(Project source, Configuration config, FileArtifact varModel) {
2
3    def main(Configuration config, FileArtifact varModel) {
4
5      <?xml version="1.0" encoding="utf-8">
6      <cocktail>
7        <variability>
8
9          for (DecisionVariable dv : config.variables()) {
10             createCocktailVariable(dv);
11          }
12        </variability>
13      </cocktail>
14    }
15
16    def createCocktailVariable(DecisionVariable variable) {
17      ' <element derived=false name="$variable.getName()"/>'
18    }
19  }
20
21 }

```

Figure 5: Cocktail variability model template in VIL.

The second template for the Cocktail resolution model is quite similar to the Cocktail variability model as shown in Figure 6. The only differences are in the `resolution`-element of the `main` definition (lines 7 and 13) and in the attributes of the `element`-elements that are created as part of the `resolution`-element (line 19). Further, assume the same name-matching between the decision variables and the names expected by Cocktail as mentioned above.



```

1  template cocktailResModel(Project source, Configuration config, FileArtifact resModel) {
2
3    def main(Configuration config, FileArtifact resModel) {
4
5      <?xml version="1.0" encoding="utf-8">
6      <cocktail>
7        <resolution>
8
9          for (DecisionVariable dv : config.variables()) {
10             createCocktailResolution(dv);
11          }
12        </resolution>
13      </cocktail>
14    }
15
16    def createCocktailResolution(DecisionVariable variable) {
17      ' <element resolves="$variable.getName()" value="$variable.getValue()" bindingTime=compile/>'
18    }
19  }
20
21 }

```

Figure 6: Cocktail resolution model template in VIL.

Using VIL, the mapping of the configuration space to the implementation space is defined and the VIL build script as well as the VIL templates can be used to create specific content-sharing application variants according to a specific configuration.

It should be noted that in the example given above, part of the instantiation is actually performed by Cocktail. However, VIL is much more powerful than this (actually quite simple) instantiator can show: it can even use arbitrary external programs to perform part of the instantiation process. Thus, it can, for example, weave aspects, use compilers or other pre-processors, etc.

4.3 Improved Configuration of a Domain-Specific Service Platform

In this section, we will adopt the perspective of a Platform Variant Creator and describe the improvements of the configuration support in EASy-Producer.

Figure 7 shows the revised **IVML Configuration Editor** of the **Product Line Editor** for the configuration of a specific instance of the content-sharing platform, the audio-sharing application (cf. Deliverable D2.4.1). Besides the new design of the decision variable table (visualization for nested elements by indentation), the following new columns ease the configuration and increase the usability:

- **+**: Adds a new element to a sequence or a set of decision variables.
- **-**: Deletes an existing element from a sequence or a set of decision variables.
- **Freeze**: Allows freezing of individual decision variables by clicking the cell of the freeze-column in the line of the desired decision variable (the concept of freezing is described in detail in the IVML Language Specification in Section 7)
- **Comment**: Displays additional text that supports the decision towards a specific configuration of a decision variable (cf. Section 4.1).

Further, the IVML Configuration Editor provides a set of new buttons that ease the configuration. The **Propagate Values** button triggers the reasoner to check whether decision variables can be automatically assigned with a value based on the values of other decision variables and the constraints defined in the variability model (cf. Section 2.1). The **Freeze All** button freezes all assigned decision variables of the current configuration. This is, in particular, helpful when it comes to large-scale variability models with a plethora of decision variables. Finally, the **Undo Changes** button allows reverting changes made to the configuration in order to correct certain assignments and actions, e.g., freezing the wrong decision variable.

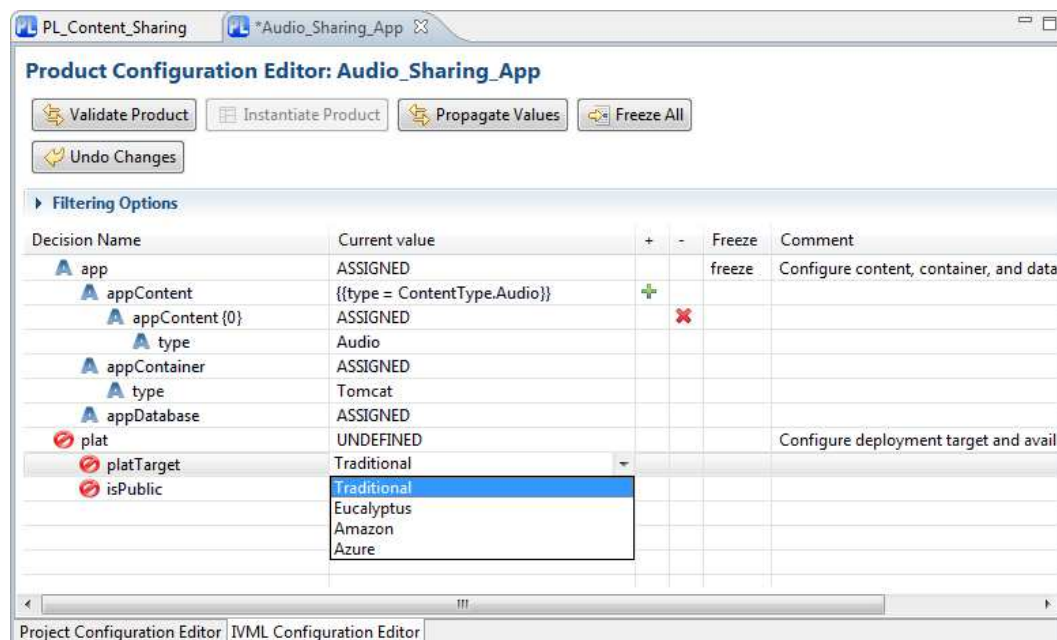


Figure 7: Product configuration using the IVML configuration editor.

5 Conclusion

This deliverable provides an overview of the current (and final) state of the INDENICA variability engineering tool set. While work has been done on various aspects since the interim deliverable, like the usability of the tool, its reasoner implementation, the configuration language IVML and others, the major invention in the final stage of the project was the introduction of the Variability Implementation Language (VIL) language, or more precisely the VIL set of languages as it includes a build language and a template language. While the prototype itself aggregates all the work done in the project so far, we did not rehash all the aspects that have already been described in previous deliverables. Instead we attach in the appendixes several guideline and specification documents, which provide a more detailed look at the extent and capabilities of the tool set.

In the previous sections we discussed the major changes we made and in particular the reasoning underlying the VIL language. Of course, many more aspects could be discussed in principle, like the various alternative approaches we discussed and tried prior to settling on the VIL-language concepts. We refrained in this deliverable from a discussion of the many VIL-language concepts; rather we would like to point to the attached VIL-language guide in the appendix for a discussion of this. We illustrated the VIL-language with a brief walk-through of its use to show that it can be used rather easily despite its extreme power. Overall VIL provides a generic transformation language for product line assets.

The Variability Engineering toolset has achieved a rather mature state, there are still various aspects, which could be improved in the future. Examples are further work in the reasoning and value propagation algorithms, further work on the tool usability, etc. Another avenue of future work is to apply it in a significant number of large-scale, real-world projects. This would produce further data regarding the benefits and drawbacks of this tool set and potential aspects of improvement.

6 References

- [1] Eclipse Foundation. Xpand, 2013. Online available at: <http://projects.eclipse.org/projects/modeling.m2t.xpand>.
- [2] Eclipse Foundation. Xtend 2.4.0 User Guide, 2013. Online available at: <http://www.eclipse.org/xtend/documentation/2.4.0/Documentation.pdf>.
- [3] Eclipse Foundation. Xtext 2.4 Documentation, 2013. Online available at: <http://www.eclipse.org/Xtext/documentation/2.4.0/Documentation.pdf>.
- [4] 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>.
- [5] The Apache Software Foundation. The Apache Velocity Project, 2013. Online available at: <http://velocity.apache.org/>.

7 Appendix: EASy-Producer Documentation

In this section, we provide the complete documentation of the Variability Engineering Tool EASy-Producer. This includes the following documents:

- **The EASy-Producer User Guide:** This guide introduces the reader to the EASy-Producer tool from the perspective of a typical user (domain engineers, application engineers, etc.). The guide starts with basic concepts supported by the tool and a step-wise description of the installation. Further, a running example will illustrate the application of EASy-Producer. Finally, a detailed description of the editors will be given.
- **The EASy-Producer Developer Guide:** This guide introduces the reader to the different extension mechanisms provided by EASy-Producer. This guide, in particular, addresses software developers or domain engineers who want to extend existing capabilities or need to define new capabilities currently not supported by the tool. EASy-Producer can be extended in terms of new instantiators, new artefact types, and new reasoners. For each of these extensions, the guide provides a general introduction to the respective concept and a step-wise description of the extension illustrated by an example.
- **The IVML Language Specification:** This document provides the complete specification of the INDENICA Variability Modelling Language (IVML), including the basic and the advanced modelling concepts, the IVML constraint language, and the language grammar.
- **The VIL Language Specification:** This document provides the complete specification of the INDENICA Variability Implementation Language (VIL). This includes the VIL Build Language, the VIL Template Language, the VIL expression language, and the grammar of the different languages.



Stiftung University of Hildesheim
Marienburger Platz 22
31141 Hildesheim
Germany



Software Systems Engineering (SSE)
Institute for Computer Science
Faculty for Mathematics, Natural
Science, Economics, and Computer
Science



EASy-Producer

Engineering Adaptive Systems

User Guide

Version 1.0

27.09.2013

Version

0.1	23.08.2012	Initial version.
0.2	10.09.2012	Table of content, initial introduction, prerequisites, and installation section added.
0.3	22.10.2012	Changes due to migration to Xtext version 2.3.1, preface added, modification and extension of Sections 1, 3 and 4. Sections 2, 5, and appendix initially added.
0.4	30.12.2012	Section 5 updated (screenshots and descriptions added).
0.5	04.03.2013	Section 3 updated (inclusion of Xtext features in EASy update site).
0.6	02.09.2013	Section 4 updated (inclusion of VIL)
1.0	27.09.2013	Version 1.0 completed (reference to other developers guide, IVML and VIL language specification, corrected spelling, updated figures)

Preface

EASy-Producer is a Software Product Line Engineering tool developed by the Software Systems Engineering (SSE) group at the University of Hildesheim.

The tool is available as an Eclipse plug-in under the terms of the Eclipse Public License (EPL) Version 1.0

The SSE group hosts the following EASy-Producer update site for easy installation and updates:

<http://projects.sse.uni-hildesheim.de/easy/>

Table of Contents

1.	Introduction.....	5
2.	Software Product Line Engineering at a Glance	6
2.1.	Basic Software Product Line Engineering.....	6
2.2.	Staged Configuration and Instantiation.....	6
2.3.	Multi Software Product Lines	6
3.	Installation	8
3.1.	Prerequisites.....	8
3.2.	Installation: Step by Step	8
3.3.	Technical Recommendations.....	10
3.4.	Further Guides and Specifications.....	10
4.	Getting Started: Product Line Engineering is EASy	12
4.1.	Running Example.....	12
4.2.	Defining a New Base Service Platform.....	13
4.2.1.	Configuration Space Definition	14
4.2.2.	Implementation Space Definition.....	16
4.3.	Deriving a Domain-Specific Service Platform.....	18
4.3.1.	Configuration of a Domain-Specific Service Platform	18
4.3.2.	Instantiation of a Domain-Specific Service Platform.....	19
5.	EASy-Producer in Detail.....	21
5.1.	The Product Line Project Structure	21
5.2.	The Product Line Editor	21
5.2.1.	The Project Configuration Editor.....	22
5.2.2.	The IVML Configuration Editor	23
A.	Appendix.....	25
A.1.	Running Example IVML-File	25
A.2.	Running Example VIL Build Script-File.....	26

1. Introduction

EASy-Producer¹ is a Software Product Line Engineering (SPLE) tool which facilitates the most recent trends and concepts in SPLE, such as large-scale Multi-Software Product Lines (MSPL), product line hierarchies, and staged configuration and instantiation. The focus of this tool is to support these rather complex concepts in an easy-to-use way. Thus, this tool allows developing a first prototypical Software Product Line (SPL) within minutes. Further, EASy-Producer is a research prototype for demonstrating new approaches to SPLE in general and, in particular approaches for simplifying the development of SPLs developed by the Software Systems Engineering group (SSE) at the University of Hildesheim.

This live-document provides a user guide that introduces the reader to the concepts and capabilities of EASy-Producer. In Section 2, we will give a brief overview on the SPLE concepts supported by EASy-Producer. This will include introductions to the concepts of SPLE in general, staged configuration and instantiation, MSPL, and product line hierarchies.

Section 3 will give guidance for the first steps in EASy-Producer. This section includes the mandatory prerequisites, the installation guide, and additional recommendations for running the tool successfully.

In Section 4, we will introduce EASy-Producer in terms of describing the development of a first prototypical SPL and the derivation of a product line product. This will cover all aspects of SPL development ranging from creating a new product line project in EASy-Producer, defining a variability model and implementing the corresponding product line artefacts, to the derivation, configuration, and instantiation of a specific product. While the purpose of this section is to describe and illustrate the basic application of EASy-Producer, we will not discuss all details of the tool at this point. This will be part of the next section.

Finally, Section 5 will describe EASy-Producer in detail. This includes detailed descriptions of the individual editors and views of the tool.

¹ EASy is an abbreviation for Engineering Adaptive Systems.

2. Software Product Line Engineering at a Glance

EASy-Producer supports basic Product Line Engineering and also staged configuration and Multi Software Product Lines or any combination of these techniques. In the next three sections we will give a short introduction to these concepts.

2.1. Basic Software Product Line Engineering

Software Product Line Engineering (SPLE) is a software development approach which focuses on the extensive reuse of artefacts involved or produced in the software lifecycle. The overall goal of SPLE is to provide a high degree of automation for the configuration and adaptation of product variants. This approach reduces the development effort and costs as well as the time-to-market while increasing the overall software quality.

A Software Product Line (SPL) is a set of related software products which are developed based on a common infrastructure but differ with respect to their provided functionalities. These differences are called variabilities.

2.2. Staged Configuration and Instantiation

Staged configuration and especially staged instantiation are approaches for facilitating partial derivation of product artefacts. These partial instantiated artefacts can still contain open variabilities while other variabilities are already bound. Thus, the configuration can be connected in arranged series. This technique can be used to support different stakeholder/user groups or to create a common basis for related sub-sets of a product line. See Figure 1 for an illustrative example.

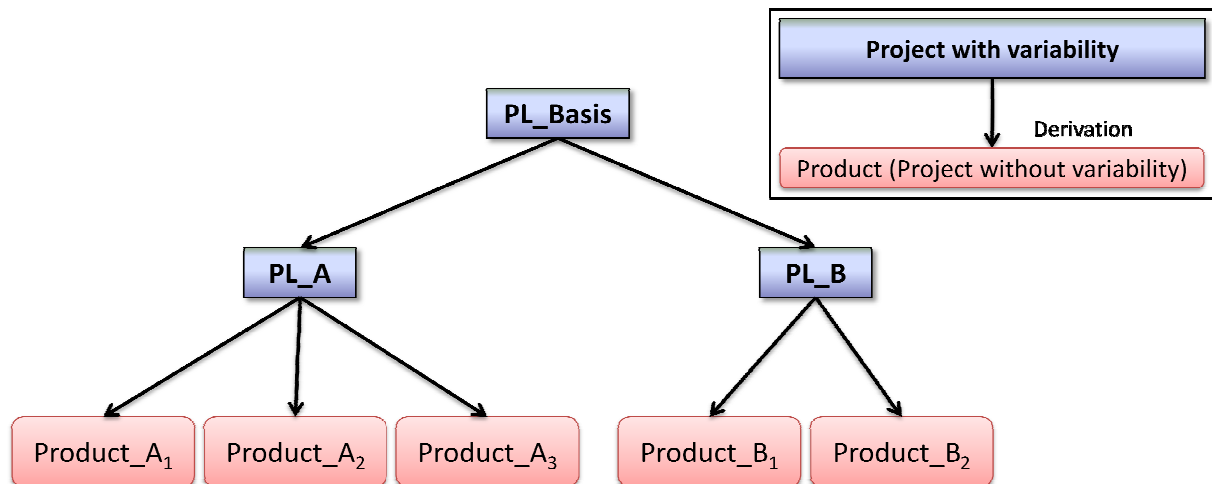


Figure 1: Example for staged configuration

2.3. Multi Software Product Lines

Multi Software Product Lines (MSPLs) are able to compose several (independent) product lines to form new products (or product lines). While forming an MSPL, the variability models of the single

product lines are combined to an integrated variability model. Derived products can contain instantiated artefacts from all combined product lines. See Figure 2 for an illustrative example.

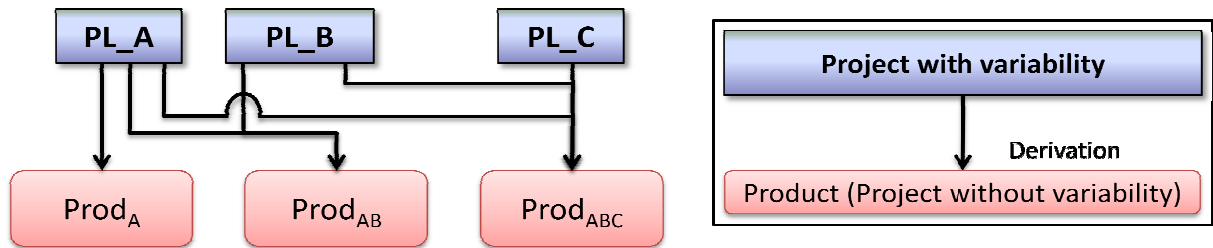


Figure 2: Example for a Multi Software Product Line.

3. Installation

In this section, we will describe the installation of EASy-Producer. In order to guarantee a successful installation, we will introduce a set of mandatory prerequisites. This will be part of Section 3.1 in which we will set up the environment in for EASy-Producer. In Section 3.2, we will describe the installation of the tool in a step-wise manner using the Eclipse update site mechanism and the EASy-Producer update site. Finally, Section 3.3 will give some technical recommendations, while Section 3.4 introduces additional guides and specifications for EASy-Producer.

3.1. Prerequisites

EASy-Producer is developed as an **Eclipse**² plug-in and requires **Xtext**³ **version 2.3.1**. Thus, in general, any Eclipse installation with Xtext version 2.3.1 is fine for installing and running EASy-Producer. However, we cannot guarantee that any combination of Eclipse and Xtext version 2.3.1 will work with EASy-Producer. Thus, we propose the following Eclipse versions as they are tested with EASy-Producer (and Xtext version 2.3.1):

- Eclipse 3.6 (Helios)
- Eclipse 3.7 (Indigo)
- Eclipse 4.0 (Juno)

We recommend using **Eclipse 3.7 (Indigo)** as this is the most exhaustively tested version of Eclipse with EASy-Producer. Download an Eclipse package from <http://www.eclipse.org/downloads/>.

Please note that Eclipse 4.2 does not work with Xtext 2.3.1 due to incompatible dependencies.

Further, Xtext version 2.3.1 has to be installed in the newly downloaded Eclipse instance. Typically, this is installed automatically when installing EASy-Producer due to plug-in dependencies. However, we encountered situations in which these dependencies were not automatically resolved. Thus, the EASy-Producer update site includes the required Xtext features. We will describe the complete installation in the next Section.

3.2. Installation: Step by Step

The SSE group hosts an EASy-Producer update site for easy installation and updates. Thus, the first step for installing EASy-Producer is to define a new update site in Eclipse. For this purpose, start Eclipse and open the *Install New Software* dialog by clicking *Help* → *Install New Software...* as shown in Figure 3:

² Eclipse website: www.eclipse.org/

³ Xtext website: <http://www.eclipse.org/Xtext/>

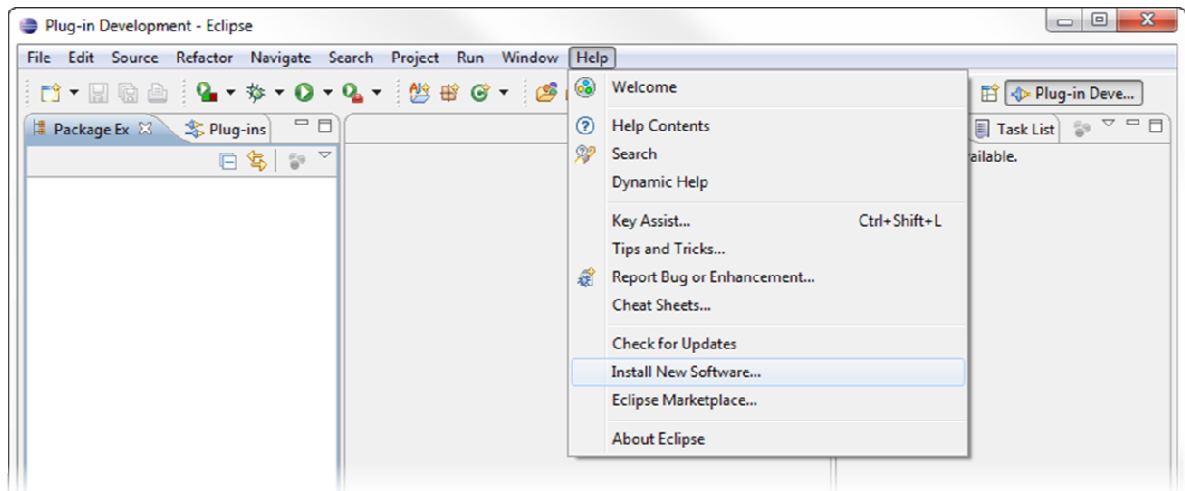


Figure 3: Open the “Install New Software” dialog

The *Install* Dialog will appear (cf. Figure 4). In this dialog, a new location for available software has to be added. Thus, click on the *Add...* button in the upper right location of the dialog.

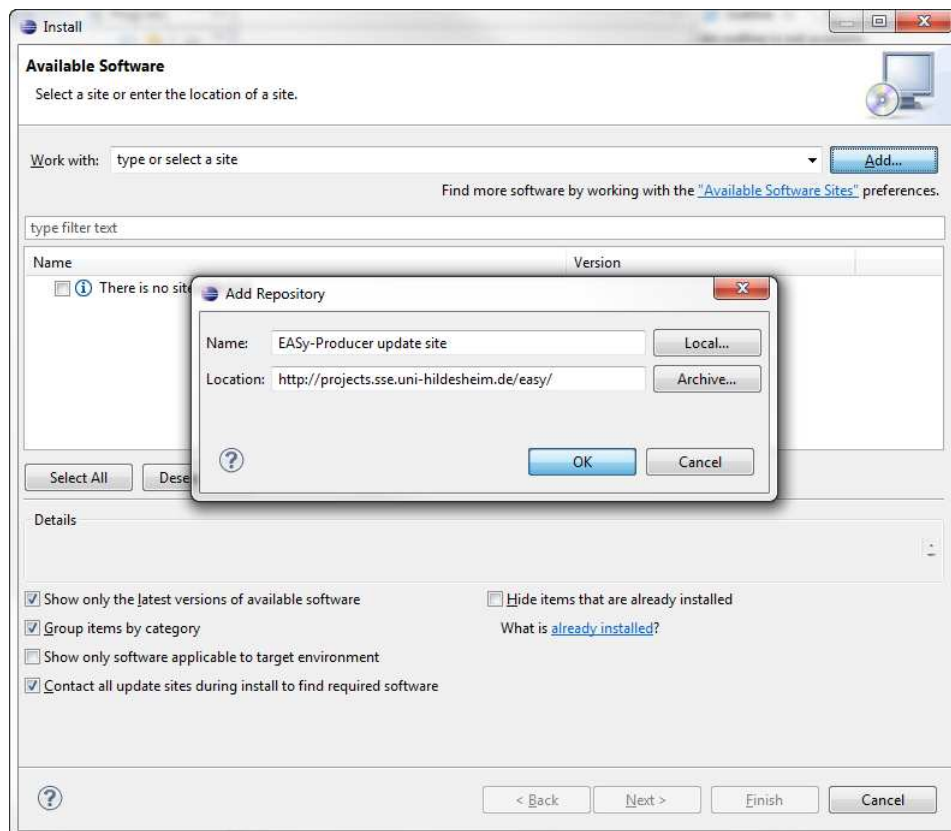


Figure 4: Add a new location for software updates

The *Add Repository* dialog requires the definition of a name for the new update site and a location as illustrated in Figure 4. The name is up to the user. For example, enter “EASy-Producer update site”. The location is the URL of the update site:

EASy-Producer update site: <http://projects.sse.uni-hildesheim.de/easy/>

Finish the definition of the new update site by clicking the OK button of the *Add Repository* dialog.

The *Install* Dialog will now contain multiple categories. If you are installing EASy-Producer for the first time and do not know which features to select, select the *Quick Installation of EASy-Producer* category. Further, select the categories *Xtend-2.3.1* and *Xtext-2.3.1* to install the required Xtext version (if not done before). This will install all required components automatically.

For more experienced users, select the categories and features as needed and click the *Next* button. Follow the steps for installing EASy-Producer (accept the license agreement and ignore the security warning for installing software of unsigned content, etc.), and restart Eclipse as prompted.

Finally, you have successfully installed the EASy-Producer.

3.3. Technical Recommendations

In order to avoid memory problems while using EASy-Producer, we recommend increasing the memory of the Eclipse application in which EASy-Producer is executed. The memory problems are due to Xtext which requires more memory than defined in a typical Eclipse configuration.

Open the “*eclipse.ini*” file in your Eclipse directory and enter the following parameters at the end of the file:

```
-vmargs  
-Xms40m  
-Xmx512m  
-XXMaxPermSize=128m
```

3.4. Further Guides and Specifications

EASy-Producer provides two expressive languages that support the creation of required software product line artefacts:

The **INDENICA Variability Modelling Language (IVML)** is an expressive, textual variability modelling language, which provides basic and advanced modelling capabilities for the definition of variability models. In order to define such a model based on IVML, we provide the IVML language specification. This specification is part of the EASy-Producer installation and can be found in the **Eclipse Help**.

The **Variability Implementation Language (VIL)** is a textual language for the flexible specification of the instantiation process of a software product line. This language consists (beside other parts) of the VIL build language and the VIL template language. The former language provides modelling elements for the specification of the individual build tasks of the instantiation process, while the latter language supports the definition of templates that can be applied to specific artefacts, for example, to manipulate their content, as part of the instantiation process. The corresponding VIL language specification is also part of the EASy-Producer installation and can be found in the **Eclipse Help**.

Further, EASy-Producer supports the extension of the tool by custom instantiations and reasoners. The **EASy-Producer Developers Guide** introduces the reader to the possible extensions and provides a step-wise description of how to extend the tool. This guide can be found in the **Eclipse Help** as well.

The EASy-Producer user guide, the EASy-Producer developers guide, as well as the IVML and the VIL language specification are also available as PDFs on the EASy-Producer update site.

4. Getting Started: Product Line Engineering is EASy

In this section, we will adopt the roles of a domain engineer and an application engineer in order to illustrate the application of EASy-Producer based on a running example. 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. Section 4.1 will describe this example in detail. In Section 4.2, we will adopt the role of a domain engineer and describe the definition of a SPL from which multiple variants of the content-sharing platform can be derived. This includes the definition of the variability model using the INDENICA⁴ Variability Modelling Language (IVML), the implementation of these variabilities in source code, and the definition of a build script for the instantiation of the generic artefacts using the Variability Implementation Language (VIL). In Section 4.3, we will adopt the role of an application engineer 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`.

Please note that we will not discuss the tool in all details in this section. This will be part of the detailed description of EASy-Producer in Section 5. Further, we will not discuss the IVML and VIL language here. A detailed description of these languages can be found in the corresponding language guides (cf. Section 3.4).

4.1. Running Example

In this section, we introduce a running example which we will use throughout Section 4 to illustrate the basic application and capabilities of EASy-Producer. In this example a content-sharing platform will be developed in terms of a SPL. A content-sharing platform allows its users to upload, annotate, release and share content of various types. In this example, concrete applications may differ with respect to:

- The supported content types such as text, video, audio, 3D content, or binary (large) objects (BLOBs).
- The hosting infrastructure which consists of a) a web container being responsible for serving the content and b) the database, which stores user and content data.
- The deployment target which may either be a traditionally hosted server, or a cloud environment. The cloud environment may be private, like a local installation of the

⁴ INDENICA is an EU-funded project in which the variability modeling language of EASy-Producer was initially designed and developed. However, this language is not INDENICA-specific but was designed with further requirements from research and industry in mind. For more information regarding INDENICA please visit the INDENICA website: <http://indenica.eu/>

Eucalyptus⁵ cloud software or public, in this example we will allow connections to Amazon⁶ or Azure⁷ cloud.

Without going into functional details of the content-sharing platform, the variabilities introduced by content types, web container, database and deployment target allow to derive a large number of different platform instances. However, some dependencies exist that restrict the selection of variants to be part of a specific platform instance. These restrictions and dependencies will be modelled in terms of constraints in the variability model in Section 4.2.1:

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

This content-sharing platform product line will be developed in the following sections using EASy-Producer. In particular, we will focus on the variability modelling, the variability implementation and the derivation of a specific platform instance.

4.2. Defining a New Base Service Platform

In this section, we will describe the process of defining the variability of a (base) service platform (a SPL) using EASy-Producer from the perspective of a domain engineer. 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. Further, we will define a corresponding build script in VIL to specify the instantiation process of the variable artefacts. The resulting base service platform (the product line project) will be the basis for the derivation of different content-sharing platforms by an application engineer.

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 (see Section 3 for installation details). Start the **New Project Wizard** by opening *File* → *New*

⁵ Eucalyptus website: <http://open.eucalyptus.com/>

⁶ Amazon cloud website: <http://aws.amazon.com/de/ec2/>

⁷ Azure website: <http://www.microsoft.com/windowsazure/>

→ **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. EASy suggests naming the newly created project with a prefix (`PL_`). However, it is not necessary to keep this prefix. 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 5.

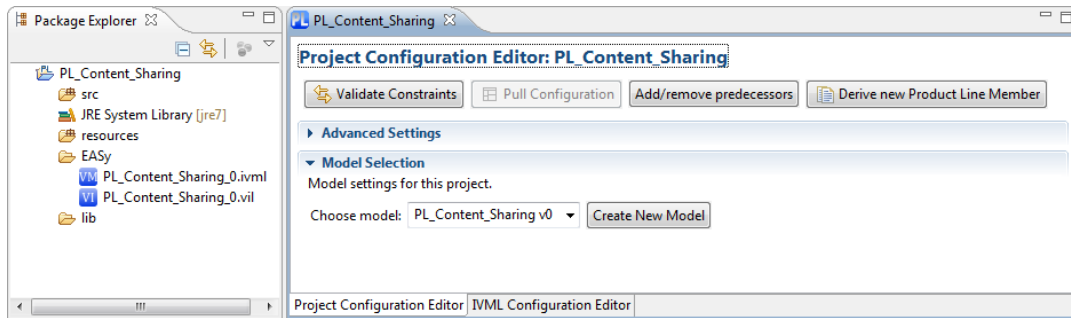


Figure 5: Running Example – The Product Line Editor.

The **Product Line Editor** is the central editor in EASy-Producer as it provides the basic information about a SPL (or a product) as well as the capabilities to derive, configure, and instantiate a product using the different tabs shown in Figure 5. 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 sections.

4.2.1. Configuration Space Definition

A variability model defines the valid configuration space of a specific SPL. The variabilities are implemented in the artefacts. In EASy-Producer, we use IVML⁸ 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 “o”). In our example, double-click the file **PL_Content_Sharing_o** 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 6. 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 “vo”.

⁸ See the IVML language specification (cf. 3.4) for a detailed description of this language.

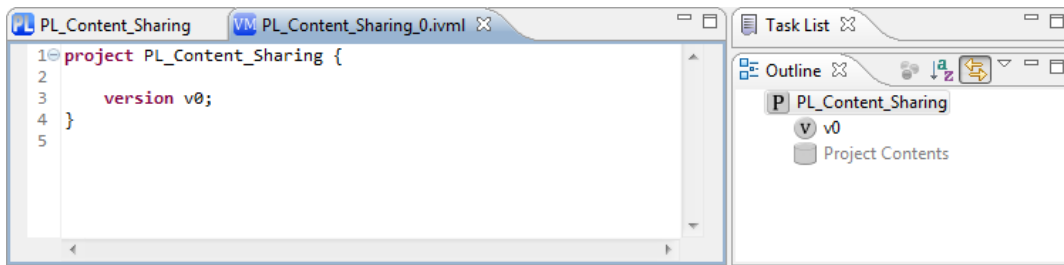


Figure 6: Running Example – The IVML Editor.

We characterize the configuration space of the variant-enabled content-sharing base platform by specifying the variability model in IVML. Figure 7 shows a snippet of the variability model (the complete model can be found in the appendix A.1). 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. Section 4.1). As indicated in the outline on the right side of Figure 7, 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 4.3.1.

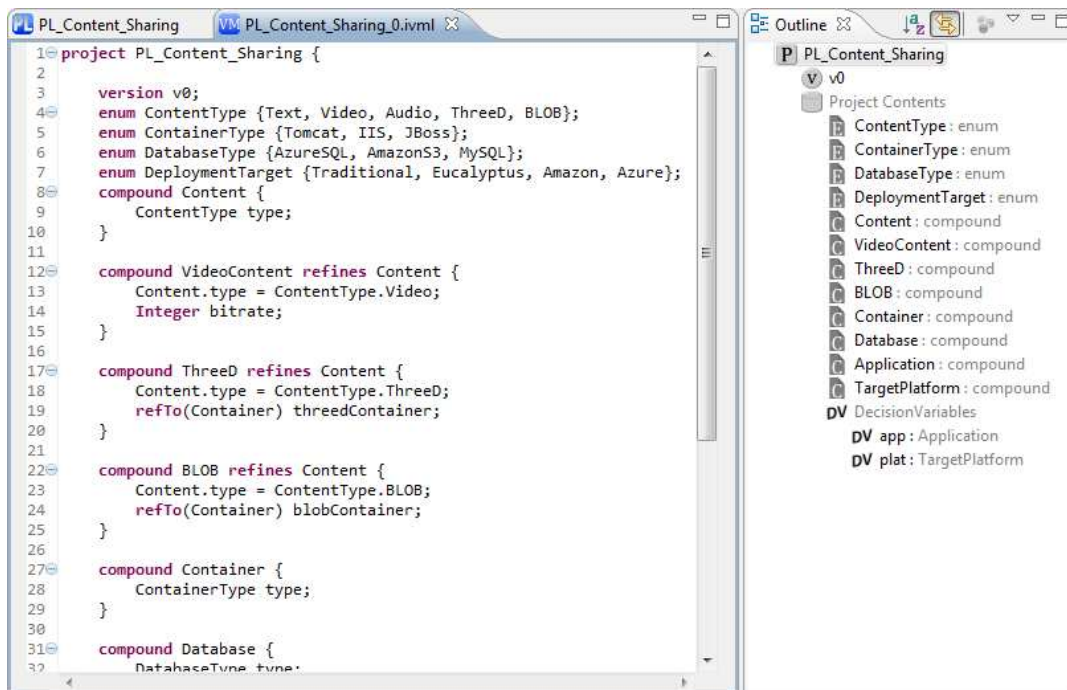
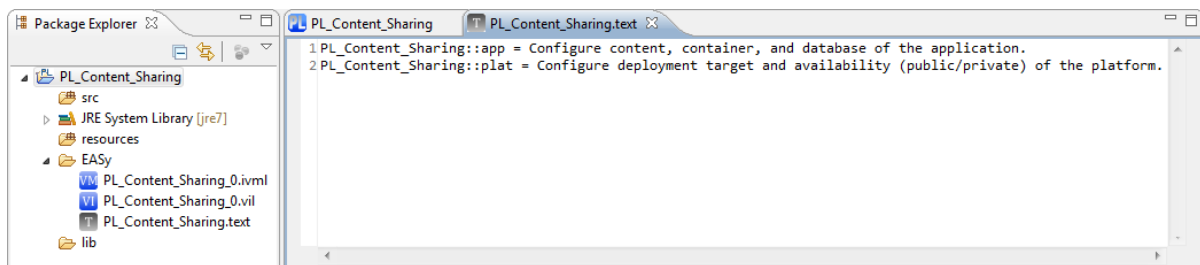


Figure 7: Running Example – The Variability Model (snippet).

In addition to the variability model, we will also define comments for the decision variables in order to support the application engineer in the configuration of a valid product. The first step is to create a new text-file in the **EASy-folder** of the product line project. For this purpose, right-click

on the **EASy-folder** and select **New** → **File**. The name of the file must match the name of the variability model (the project-name in IVML) followed by the version number to unambiguously link the comments to the desired decision variables. Thus, enter `PL_Content_Sharing_0.text` as the name of the new file and click the *Finish* button. We will open the new text-file with a simple text-editor to enter the comments. Each comment-definition will start with the name of the variability model (the project name) followed by “::”, the name of the decision variable for which we want to define the comment, and an equal-sign (“=”). The actual comment is defined in plain text after the “=”.

The result of the comment-definition is shown in Figure 8. We defined two comments for the two decision variables “app” and “plat” of the running example (cf. Figure 8a)), which are displayed in the **IVML Configuration Editor of the Product Line Editor** (cf. Figure 8b)).



a) Comment-definition in the text-file.

Filtering Options					
Decision Name	Current value	+	-	Freeze	Comment
app	UNDEFINED				Configure content, container, and database of the application.
plat	UNDEFINED				Configure deployment target and availability (public/private) of the platform.

b) Decision variables and comments in the IVML configuration editor

Figure 8: Running Example - Definition of Decision Variable Comments.

Finally, the variability model, and, thus, the configuration space of the content-sharing application is defined. We will use this model in Section 4.3.1 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.

4.2.2. Implementation Space Definition

The implementation space of a specific SPL 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 (VITs). A VIT is a specific approach to realize variability, e.g., using pre-processor directives, aspects, or any other techniques. In EASy-Producer different VITs can be applied and combined. Their application and combination is defined in a VIL⁹ build script. However, some VITs may be realized by an individual instantiator, which actually applies the VIT. A detailed discussion on the concept of instantiators in EASy-Producer can be found in the EASy-Producer developers guide (cf. Section 3.4). In the running

⁹ See the VIL language specification (cf. 3.4) for a detailed description of this language.

example, we will use the Velocity instantiator as it is one of the default instantiators of the basic EASy-Producer installation.

All product line (and product) source code is located in the **src** folder of the product line project as shown in Figure 9. The Velocity instantiator provides pre-processor functionality to Java and can be applied in terms of adding Velocity-specific statements to plain Java code. In lines 5 and 6 of Figure 9, the deployment platform and the public switch will be defined accordingly to the values of `platTarget` and the `isPublic` variables (cf. the variability model in Figure 7). Both variables are nested variables of the platform variable `plat`. Thus, they are accessed using “-”-notation. In order to guarantee that Velocity will find these variables, the instantiator requires a dollar-sign in front of the variable declarations in the code.



Figure 9: Running Example – The Variability Implementation (snippet).

The next step is to define the VIL build script. Open the **VIL Build Language Editor** by double clicking the **VIL-file** in the **EASy-folder**. The file has the name `PL_Content_Sharing_0.vil`. Figure 10 shows the VIL build script of this example (this build script can also be found in the appendix A.2). This script is rather simple: the first rule `clean` deletes all source artefacts in the target project (the product project) to guarantee that only the most recent instantiated artefacts are present in the final product. The second rule `main` is the entry-point for the VIL engine. This rule defines `clean` to be a precondition (guaranteeing that the target will be cleaned before the actual instantiation). The only action `main` defines is the call of the Velocity instantiator with the path to the generic source artefacts, the target path, and the configuration as parameters.

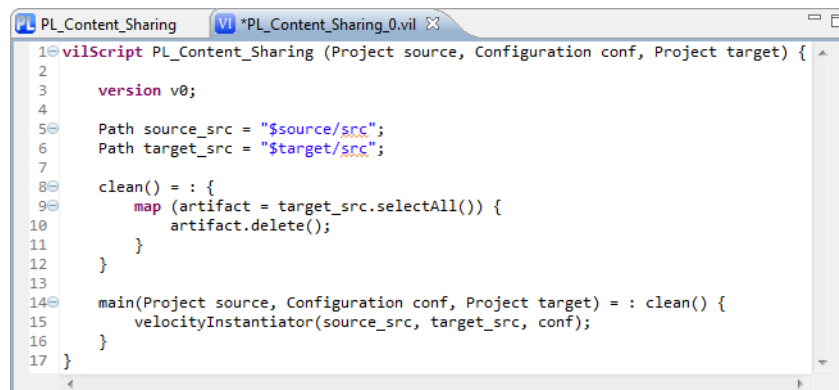


Figure 10: Running Example – The VIL Build Script.

Finally, the implementation space and the corresponding build script are defined 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.

4.3. 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 an application engineer and start with the derivation of a new product line member¹⁰ (in this case, the product project), configure the product based on the variability model defined in Section 4.2.1, 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 **Project Configuration 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 11.

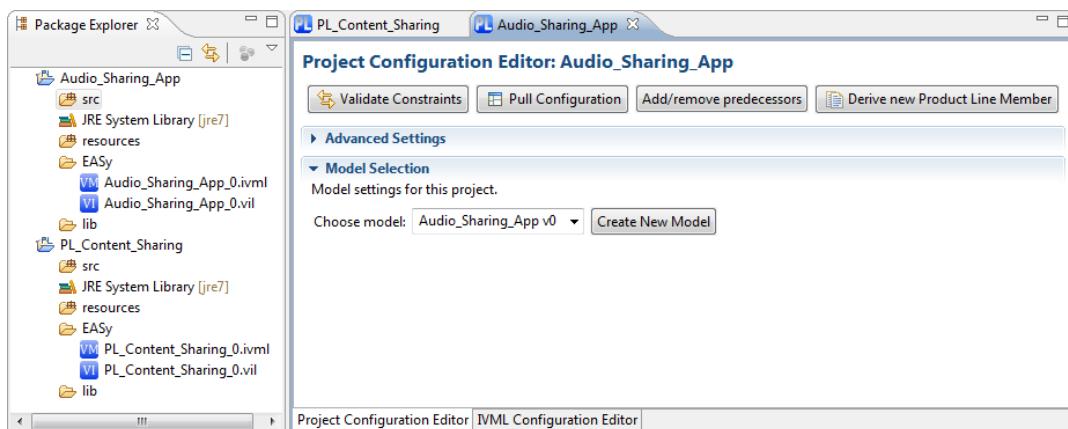


Figure 11: Running Example – The Product Derivation.

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.

4.3.1. 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.

¹⁰ 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).

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 5.2.2). Figure 12 illustrates the **IVML Configuration Editor**, including the configurable elements of our audio content-sharing application.

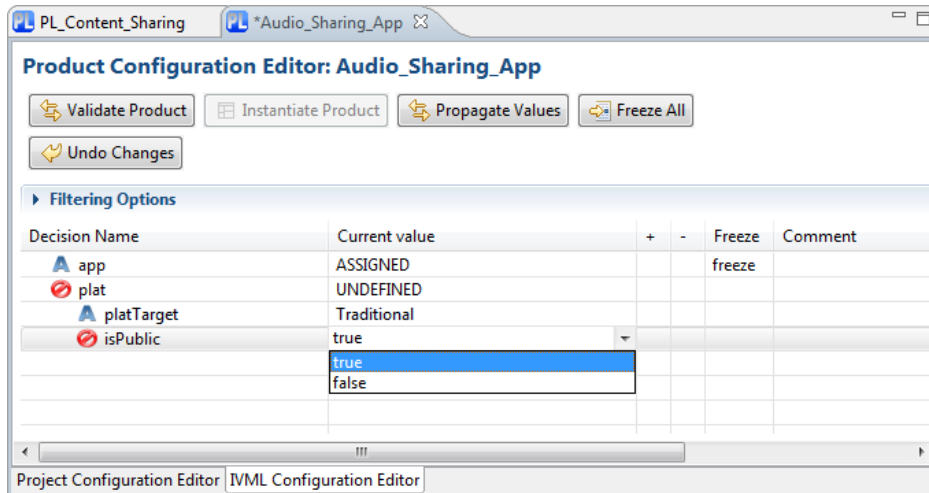


Figure 12: Running Example – The IVML Configuration Editor.

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.

Finally, the product is configured and ready for instantiation.

4.3.2. 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 VIL build script. We defined this information in the previous sections, such as the implementation space and build script (cf. Section 4.2.2) and the product configuration (cf. Section 4.3.1). This relies in turn on the configuration space definition (cf. Section 4.2.1). Thus, the last step is to click the *Instantiate Product* button in the **IVML**

Configuration Editor. This will yield the instantiated artefacts from the product line project and inserts them into the product project while resolving the variabilities.

5. EASy-Producer in Detail

In this section, we will describe EASy-Producer in detail. This includes the description of the product line project structure in Section 5.1 as well as the different editors in Section 5.2.

5.1. The Product Line Project Structure

In this section, we will discuss the product line project structure of EASy-Producer. The basic structure of each product line project equals the general structure of Java-project in Eclipse. The only difference is in the EASy-folder of the product line project. This folder contains all EASy-Producer files. These files are:

File Icon	Description
-----------	-------------



The IVML-file, which contains the variability model described in the INDENICA Variability Modelling Language, or a specific configuration.
This file is **mandatory** and will be automatically created if a new product line project is created.



The text-file, which contains additional comments for the decision variables defined in the variability model. Please note that we use a “T” for “Text” instead of a “C” for “Comments” as this may be confused with “Configuration”.
This file is **optional** and has to be created manually.



The VIL build script file, which contains the specification of the instantiation process of the variable artefacts of the product line project.
This file is **mandatory** and will be automatically created if a new product line project is created.



The VIL template file, which contains the definition of generic templates that can be applied during the instantiation process to create or manipulate specific artefacts and their content.
This file is **optional** and has to be created manually.

All files introduced above can be created manually (also those that are mandatory) by clicking *File* → *New* → *Other*. In the wizard, open the **EASy-Producer**-folder and select the file you want to create. Please note that we recommend adding such files to the **EASy-folder** of the product line project as this is the default folder for EASy-specific files. Further, the creation of a variability model, a build script, and templates are supported by individual (text) editors that will open by simply double-clicking the respective file in the **EASy-folder**.

5.2. The Product Line Editor

The **Product Line Editor** is the central editor in EASy-Producer as it provides the basic information about a SPL (or a product) as well as the capabilities to derive, configure, and instantiate a product using the different sub-editors (tabs). This editor opens automatically if a new product line project is created. In order to open the editor manually, *right click* on the product line project and select **Edit Product Line**.

5.2.1. The Project Configuration Editor

The **Project Configuration Editor** provides the general configuration options of a product line project as well as the general actions that can be performed.

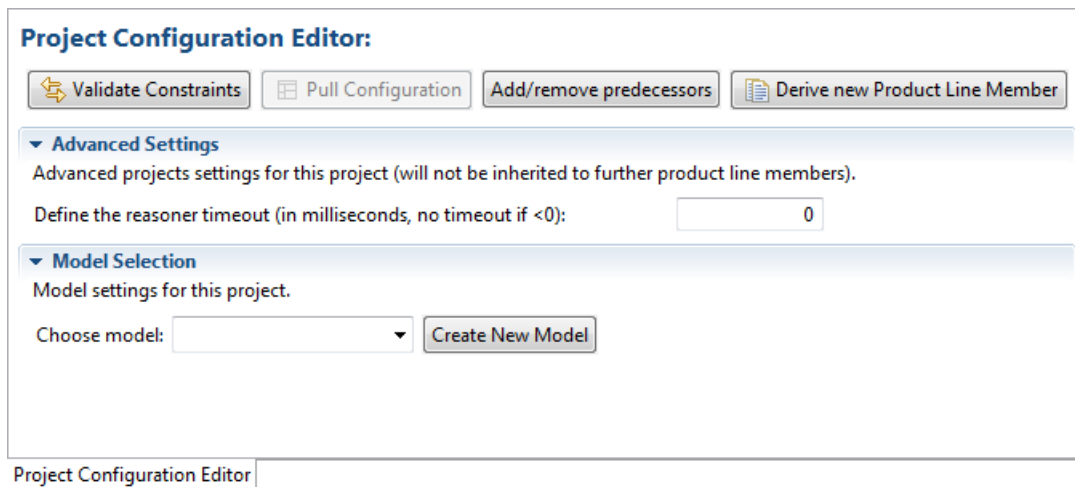


Figure 13: Project Configuration Editor.

Editor Element	Description
“Validate Constraint” Button	Validates all constraints in the selected variability model using the reasoner.
“Pull Configuration” Button	In case that a new predecessor is added (see below) the configuration of the predecessor is integrated into the configuration of the current product line project.
“Add/remove predecessor” Button	In a Multi Software Product Line scenario (cf. Section 2.3), a new product line project, e.g. a product line product, is initially derived from a single parent product line project, e.g. the base product line. In order to integrate additional parents, this button will open a dialog to select the desired product line projects of the current workspace in Eclipse. Please note that the addition of new parent product line projects also requires the Pull Configuration action (see above) to integrate the configurations of the selected parents into the current product line project.
“Derive new Product Line Member” Button	Derives a new product line project based on the current product line project. For example, this creates a new product project based on the base product line project.
Advanced Settings (Reasoner Timeout)	Restricts the time for the reasoner to calculate the validity of a specific configuration to the defined time in milliseconds.

Model Selection

In case that multiple variability models are available, this option enables the selection of the desired variability model as the basis for the configuration of the current product line project.

5.2.2. The IVML Configuration Editor

The IVML Configuration Editor supports the configuration of individual products or partially configured product lines by providing a graphical user interface for the assignment of values to decision variables defined in the variability model. This editor is also used to start the instantiation process after the configuration of a specific product.

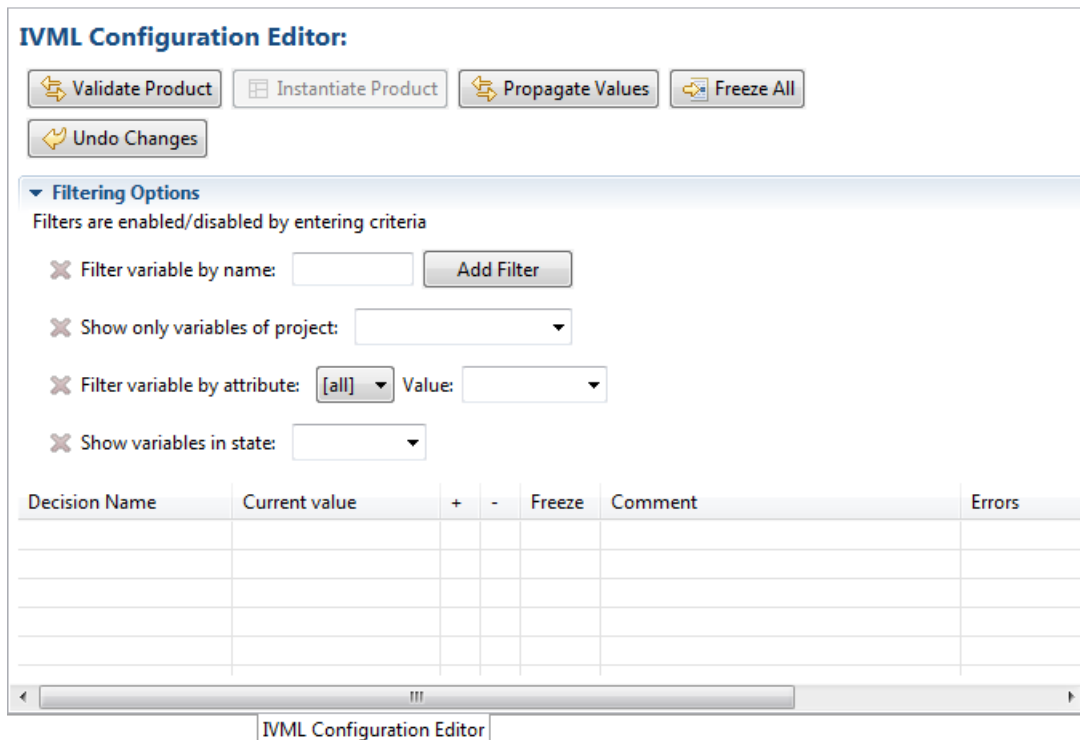


Figure 14: IVML Configuration Editor.

Editor Element

Description

“Validate Product” Button

Validates the current configuration of this product line project using the reasoner.

“Instantiated Product” Button

Instantiates the current project line project (partial product line or product) based on the current (partial) configuration.

“Propagate Values” Button

Assigns currently unassigned decision variables of the configuration automatically. This automation requires the assignment of a subset of the available decision variables and the relation of these variables to the unassigned variables in terms of constraints

“Freeze All” Button	in the variability model. Freezes all assigned decision variables of the current configuration. The values of frozen decision variables cannot be altered afterwards. For details on the concept of freezing, see the IVML language specification (cf. Section 3.4).
“Undo Changes” Button	Reverts all changes since the last saving of the current configuration.
Filtering Options (Filter by name)	Filters the available decision variables of the current configuration by name.
Filtering Options (Filter by project)	Filters the available decision variables of the current configuration by the project in which they are defined.
Filtering Options (Filter by attribute)	Filters the available decision variables of the current configuration by an attribute and the specific attribute value. For details on the concept of attributes, see the IVML language specification (cf. Section 3.4).
Filtering Options (Filter by state)	Filters the available decision variables of the configuration by their state. The available states are: unassigned, assigned, and frozen.
“Decision Name” Column	The name of the decision variable.
“Current value” Column	The current value of the decision variable.
“+” Column	Adds a new element to a sequence or a set of decision variables.
“-“ Column	Deletes an existing element from a sequence or a set of decision variables.
“Freeze” Column	Freezes the decision variable in the row of the current cell.
“Comment” Column	Displays additional information regarding the purpose and the configuration options of the decision variable. This requires the definition of a text-file (cf. Section 4.2.1).
“Errors” Column	Displays errors, for example the violation of a constraint of the variability model by the current value of the decision variable in that row. This requires the validation of the current configuration.

A. Appendix

A.1. Running Example IVML-File

```
project PL_Content_Sharing {  
  
  version v0;  
  enum ContentType {Text, Video, Audio, ThreeD, BLOB};  
  enum ContainerType {Tomcat, IIS, JBoss};  
  enum DatabaseType {AzureSQL, AmazonS3, MySQL};  
  enum DeploymentTarget {Traditional, Eucalyptus, Amazon, Azure};  
  compound Content {  
    ContentType type = ContentType.Audio;  
  }  
  
  compound VideoContent refines Content {  
    Integer bitrate;  
    Content.type = ContentType.Video;  
  }  
  
  compound Container {  
    ContainerType type = ContainerType.Tomcat;  
  }  
  
  compound Database {  
    DatabaseType type = DatabaseType.MySQL;  
  }  
  
  compound Application {  
    Content appContent;  
    Container appContainer;  
    Database appDatabase;  
  }  
  
  compound TargetPlatform {  
    DeploymentTarget platTarget;  
    Boolean isPublic;  
  }  
  
  compound ThreeD refines Content {  
    refTo(Container) threedContainer;  
    Content.type = ContentType.ThreeD;  
  }  
  
  compound BLOB refines Content {  
    refTo(Container) blobContainer;  
    Content.type = ContentType.Audio;  
  }  
  
  Application app;  
  TargetPlatform plat;  
  
  plat.platTarget == DeploymentTarget.Traditional implies  
    app.appDatabase.type == DatabaseType.MySQL;  
  plat.platTarget == DeploymentTarget.Eucalyptus implies  
    app.appDatabase.type == DatabaseType.AmazonS3;
```

```
plat.platTarget == DeploymentTarget.Amazon implies
  app.appDatabase.type == DatabaseType.AmazonS3;
plat.platTarget == DeploymentTarget.Azure implies
  app.appDatabase.type == DatabaseType.AzureSQL;
}
```

A.2. Running Example VIL Build Script-File

```
vilScript PL_Content_Sharing (Project source, Configuration conf, Project target)
{
    version v0;

    Path source_src = "$source/src";
    Path target_src = "$target/src";

    clean() = : {
        map (artifact = target_src.selectAll()) {
            artifact.delete();
        }
    }

    main(Project source, Configuration conf, Project target) = : clean() {
        velocityInstantiator(source_src, target_src, conf);
    }
}
```



Stiftung University of Hildesheim
Marienburger Platz 22
31141 Hildesheim
Germany



Software Systems Engineering (SSE)
Institute for Computer Science
Faculty for Mathematics, Natural
Science, Economics, and Computer
Science



EASy-Producer

Engineering Adaptive Systems

Developers Guide

Version 1.0

20.09.2013

Version

0.1	28.08.2012	Initial version
0.2	10.09.2012	Reasoning section moved to the end of the document, prerequisite and installation added, debug flags added to section 3
0.3	04.12.2012	Preface added, Section 3.1.1 added
0.4	04.03.2013	Instantiator and reasoning section updated
0.5	01.04.2013	General corrections, e.g. spelling.
0.6	15.08.2013	Initial inclusion of VIL
1.0	20.09.2013	VIL section updated, according updates to the instantiator section

Preface

EASy-Producer is a Software Product Line Engineering tool developed by the Software Systems Engineering (SSE) group at the University of Hildesheim.

The tool is available as an Eclipse plug-in under the terms of the Eclipse Public License (EPL) Version 1.0

The SSE group hosts the following EASy-Producer update site for easy installation and updates:

<http://projects.sse.uni-hildesheim.de/easy/>

Table of Contents

1.	Introduction.....	5
2.	Installation	6
2.1.	Prerequisites.....	6
2.2.	Installation: Step by Step	6
2.3.	Technical Recommendations.....	8
2.4.	Further Guides and Specifications.....	8
3.	EASy-Producer Extensions.....	10
3.1.	Implementing a New Instantiator	10
3.1.1.	Instantiation Concept in EASy-Producer	11
3.1.2.	Eclipse Plug-in Project Creation and Configuration for New Instantiators	14
3.1.3.	Instantiator Implementation	18
3.1.4.	Instantiator Integration	20
3.2.	Implementing a New VIL Artefact Type.....	21
3.2.1.	The VIL Artefact Model in EASy-Producer.....	21
3.2.2.	Eclipse Plug-in Project Creation and Configuration for New Artefacts	23
3.2.3.	Artefact Implementation	23
3.3.	Implementing a New Reasoner	28
3.3.1.	Eclipse Plug-in Project Creation and Configuration for New Reasoners.....	28
3.3.2.	Reasoner Implementation	29
3.3.3.	Reasoner Integration	32

1. Introduction

EASy-Producer¹ is a Software Product Line Engineering (SPLE) tool which facilities the most recent trends and concepts in SPLE, such as large-scale Multi-Software Product Lines (MSPL), product line hierarchies, and staged configuration and instantiation. The focus of this tool is to support these rather complex concepts in an easy-to-use way. Thus, this tool allows developing a first prototypical Software Product Line (SPL) within minutes. Further, EASy-Producer is a research prototype for demonstrating new approaches to SPLE in general and, in particular approaches for simplifying the development of SPLs developed by the Software Systems Engineering group (SSE) at the University of Hildesheim.

This live-document provides a developers guide that introduces the reader to the basic capabilities of EASy-Producer and how to develop further extensions to this tool. In Section 2, we will give guidance for the first steps with EASy-Producer. This section includes the mandatory prerequisites, the installation guide, and additional recommendations for running the tool successfully. This also provides the development environment, in which extensions for EASy-Producer will be created.

Section 3 will describe the different extension mechanisms of EASy-Producer. This includes the implementation of new instantiators, new artefact types, and new reasoners. For each of these extensions, we will briefly introduce the basic concepts and provide a step-wise example of how to create a new extension of the specific type.

¹ EASy is an abbreviation for Engineering Adaptive Systems.

2. Installation

In this section, we will describe the installation of EASy-Producer. In order to guarantee a successful installation, we will introduce a set of mandatory prerequisites. This will be part of Section 2.1 in which we will set up the environment for EASy-Producer. In Section 2.2, we will describe the installation of the tool in a step-wise manner using the Eclipse update site mechanism and the EASy-Producer update site. Finally, Section 2.3 will give some technical recommendations, while Section 2.4. introduces additional guides and specifications for EASy-Producer.

2.1. Prerequisites

EASy-Producer is developed as an **Eclipse**² plug-in and requires **Xtext**³ **version 2.3.1**. Thus, in general, any Eclipse installation with Xtext version 2.3.1 is fine for installing and running EASy-Producer. However, we cannot guarantee that any combination of Eclipse and Xtext version 2.3.1 will work with EASy-Producer. Thus, we propose the following Eclipse versions as they are tested with EASy-Producer (and Xtext version 2.3.1):

- Eclipse 3.6 (Helios)
- Eclipse 3.7 (Indigo)
- Eclipse 4.0 (Juno)

We recommend using **Eclipse 3.7 (Indigo)** as this is the most exhaustively tested version of Eclipse with EASy-Producer. Download an Eclipse package from <http://www.eclipse.org/downloads/>.

Please note that Eclipse 4.2 does not work with Xtext 2.3.1 due to incompatible dependencies.

Further, Xtext version 2.3.1 has to be installed in the newly downloaded Eclipse instance. Typically, this is installed automatically when installing EASy-Producer due to plug-in dependencies. However, we encountered situations in which these dependencies were not automatically resolved. Thus, the EASy-Producer update site includes the required Xtext features. We will describe the complete installation in the next Section.

2.2. Installation: Step by Step

The SSE group hosts an EASy-Producer update site for easy installation and updates. Thus, the first step for installing EASy-Producer is to define a new update site in Eclipse. For this purpose, start Eclipse and open the *Install New Software* dialog by clicking *Help* → *Install New Software...* as shown in Figure 1:

² Eclipse website: www.eclipse.org/

³ Xtext website: <http://www.eclipse.org/Xtext/>

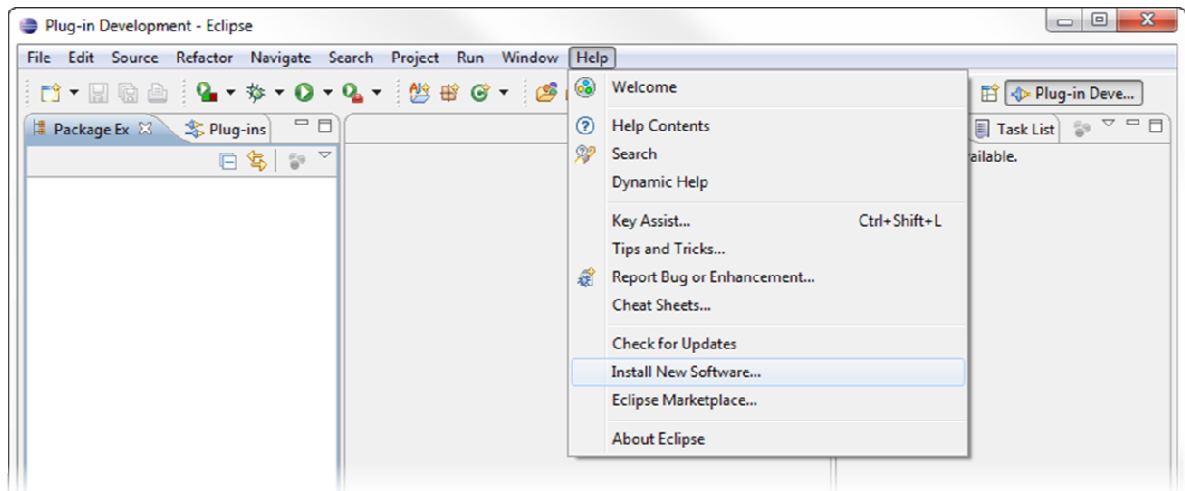


Figure 1: Open the “Install New Software” dialog

The *Install* Dialog will appear (cf. Figure 2). In this dialog, a new location for available software has to be added. Thus, click on the *Add...* button in the upper right location of the dialog.

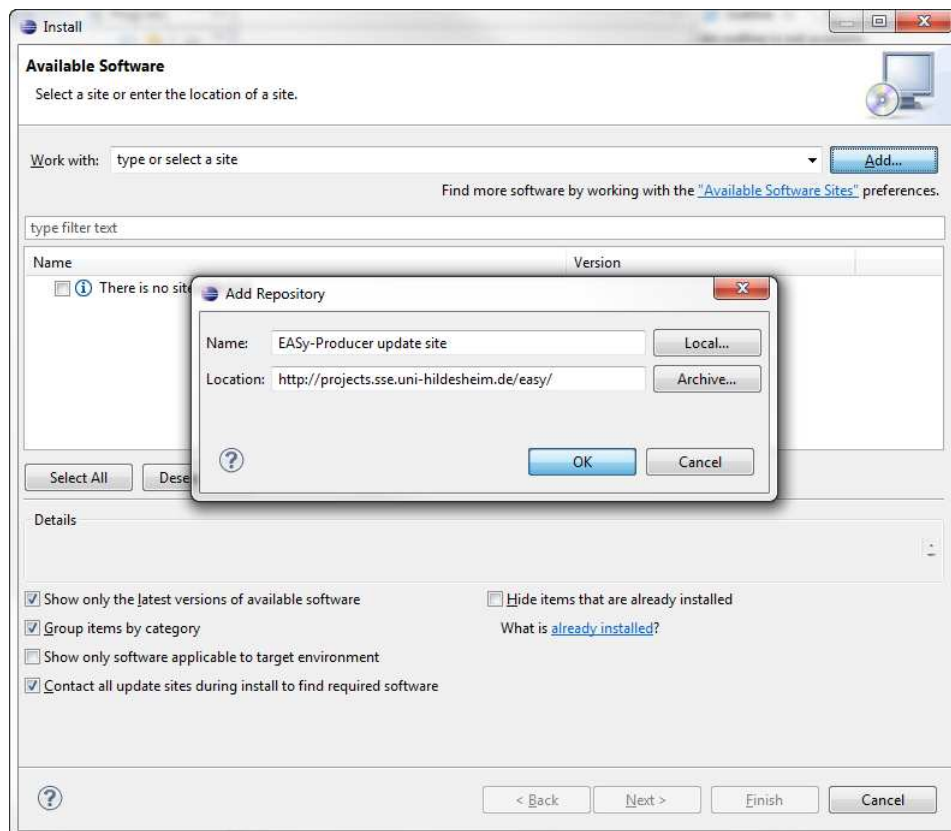


Figure 2: Add a new location for software updates

The *Add Repository* dialog requires the definition of a name for the new update site and a location as illustrated in Figure 2. The name is up to the user. For example, enter “EASy-Producer update site”. The location is the URL of the update site:

EASy-Producer update site: <http://projects.sse.uni-hildesheim.de/easy/>

Finish the definition of the new update site by clicking the OK button of the *Add Repository* dialog.

The *Install* Dialog will now contain multiple categories. If you are installing EASy-Producer for the first time and do not know which features to select, select the *Quick Installation of EASy-Producer* category. Further, select the categories *Xtend-2.3.1* and *Xtext-2.3.1* to install the required Xtext version (if not done before). This will install all required components automatically.

For more experienced users, select the categories and features as needed and click the *Next* button. Follow the steps for installing EASy-Producer (accept the license agreement and ignore the security warning for installing software of unsigned content, etc.), and restart Eclipse as prompted.

Finally, you have successfully installed the EASy-Producer.

2.3. Technical Recommendations

In order to avoid memory problems while using EASy-Producer, we recommend increasing the memory of the Eclipse application in which EASy-Producer is executed. The memory problems are due to Xtext which requires more memory than defined in a typical Eclipse configuration.

Open the “*eclipse.ini*” file in your Eclipse directory and enter the following parameters at the end of the file:

```
-vmargs  
-Xms40m  
-Xmx512m  
-XXMaxPermSize=128m
```

2.4. Further Guides and Specifications

EASy-Producer provides two expressive languages that support the creation of required software product line artefacts:

The **INDENICA Variability Modelling Language (IVML)** is an expressive, textual variability modelling language, which provides basic and advanced modelling capabilities for the definition of variability models. In order to define such a model based on IVML, we provide the IVML language specification. This specification is part of the EASy-Producer installation and can be found in the **Eclipse Help**.

The **Variability Implementation Language (VIL)** is a textual language for the flexible specification of the instantiation process of a software product line. This language consists (beside other parts) of the VIL build language and the VIL template language. The former language provides modelling elements for the specification of the individual build tasks of the instantiation process, while the latter language supports the definition of templates that can be applied to specific artefacts, for example, to manipulate their content, as part of the instantiation process. The corresponding VIL language specification is also part of the EASy-Producer installation and can be found in the **Eclipse Help**.

Further, EASy-Producer provides a user guide, which introduces the reader to the basic concepts and the different capabilities of the tool. The **EASy-Producer User Guide** can be found in the **Eclipse Help** as well.

The EASy-Producer user guide, the EASy-Producer developers guide, as well as the IVML and the VIL language specification are also available as PDFs on the EASy-Producer update site.

3. EASy-Producer Extensions

EASy-Producer provides an extension point mechanism to add additional functionality to the basic implementation. An extension is always implemented as an Eclipse plug-in and may provide customer-specific functionalities in terms of individual instantiators, artefact types, or reasoners. Custom instantiators may be capable of instantiating artefacts of different types or in a specific way. Artefact types will enable the definition and manipulation of specific artefacts as part of the instantiation process. A new reasoner may provide new or adapted capabilities to check, for example, whether a variability model or a specific product configuration is valid. In order to ease the development and integration of such extensions, EASy-Producer is capable of automatically searching and integrating new plug-ins through Eclipse Dynamic Services⁴. Thus, developers only have to provide the necessary information to EASy-Producer to include their desired functionalities.

In this section, we will describe how to implement extensions to the EASy-Producer tool. In Section 3.1, we will describe the implementation of a new instantiator and its integration in EASy-Producer. Section 3.2 will describe the implementation and integration of a new artefact type, while in Section 3.3, we will implement and integrate a new reasoner. Each section will provide detailed guidance from project creation and configuration to the final integration of the custom plug-ins in EASy-Producer.

In order to debug errors and failures during the development of EASy-Producer extensions, add the following flags to the “Run Configuration” of your Eclipse as needed (introduce the new flags with a single prefixed “-D” in the Run Configuration):

- **-debug:** This flag will print information on the variability model of EASy-Producer
- **-log:** This flag will print EASy-internal debug messages, such as errors, etc.
- **-equinox.ds.debug:** This flag will print debug messages regarding the service registration mechanism. For more details, see Section 3.1.2.
- **-equinox.ds.print:** This flag will print additional information regarding the service registration mechanism. For more details, see Section 3.1.2.

Please note that the above flags are optional. They are not a prerequisite for creating extensions for EASy-Producer but may help searching and correcting errors.

3.1. Implementing a New Instantiator

An instantiator is an external and maybe third-party tool that processes product line artefacts in its specific way. For example, the Velocity instantiator, which is shipped as a default instantiator with EASy-Producer, resolves Velocity-specific tags within Java code in accordance to a specific configuration⁵. This resolution capability allows deriving individual product variants based on the configuration values and the corresponding manipulation of Java code. However, the default instantiators of EASy-Producer may be insufficient in some situations. Further, in some situations it is the better choice to realize a proper integration, e.g., if a legacy executable is used for

⁴ For more information visit: <http://eclipse.org/equinox/>

⁵ Details on the Velocity instantiator can be found in the EASy-Producer user guide (cf. Section 2.4).

instantiation (this may be called directly from VIL) and the modified artefacts shall be passed back to VIL (this is not generically supported). Thus, we provide a simple extension mechanism for integrating custom instantiators with EASy-Producer.

In the first part of this section, we will introduce the basic instantiation concept of EASy-Producer to form a common understanding of how an instantiator works. In the second part, we will describe how to set-up a new plug-in project in Eclipse for implementing a custom instantiator. This also includes the specific configurations that have to be done to utilize the automated search and integration mechanism provided by Eclipse Dynamis Services. The third part will discuss the methods that are required when implementing a new instantiator. The focus of this part will be on how, when and why EASy-Producer invokes specific methods of an instantiator. In the fourth part, we will finally show how to integrate a new instantiator.

3.1.1. Instantiation Concept in EASy-Producer

In this section, we will introduce the basic instantiation concept in EASy-Producer in order to describe how the instantiators work. In the first part, we will have a black-box view on a generic instantiator for identifying the required input (prerequisites) for an instantiator. Please note that the generic instantiator is not related to any specific variability implementation technique (VIT). Thus, we can only give a very simple view on the instantiators in general. In the second part, we will relate the identified prerequisites in terms of giving a white-box view on the generic instantiator. However, the actual logic that defines how to process artefacts depends on the used VIT. Thus, this view is again simplified.

The concept of instantiators are closely related to the Variability Implementation Language (VIL) for specifying the individual build tasks of an instantiation process. From the perspective of VIL, instantiators are reusable, black-box components that may be called as part of a specific rule in an VIL build script or as part of an VIL template. Please note that we will only discuss those parts of VIL in this guide that are relevant to the actual implementation of an instantiators (and new artefact types in Section 3.2). For further details about the language concepts, the available types, and their application, please consider the VIL language specification (cf. Section 2.4).

An instantiator in EASy-Producer in general takes a set of possibly different input and produces a set of output. Typically, the input consists of a configuration based on an IVML variability model, generic artefacts (i.e. the artefacts of a software product line including variation points, etc.), and different variants that can be applied to the variation points of the generic artefacts. Please note that the presents as well as the location of generic artefacts, variation points and variants depends on the used VIT (we will detail this below). Based on this input, the instantiator produces an instantiator- or VIT-specific output, typically, a set of product-specific artefacts with resolved variability as illustrated by Figure 3.

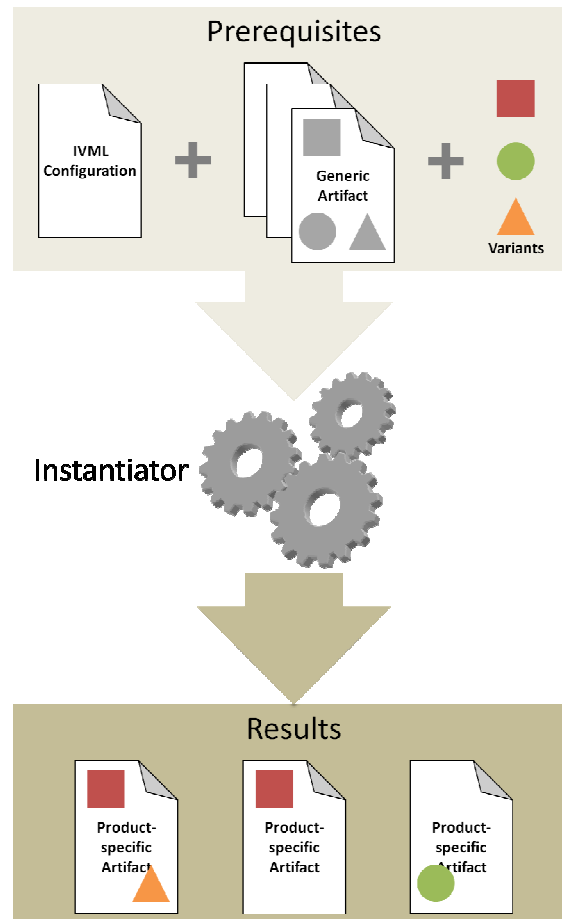


Figure 3: Black-box view of a generic instantiator (simplified)

Below, we will describe the required input (prerequisites) of an instantiator in detail:

- IVML Configuration:** The IVML configuration is based on the previously defined variability model using the IVML modeling language (explicit prerequisite not mentioned in Figure 3). A configuration includes all variable-value pairs, which are valid with respect to the constraints defined in the model. The validity of such a configuration is automatically checked before the instantiation process. This prevents from calling the instantiator with an invalid configuration, which will yield corrupted product-specific artefacts. For instantiation, only configured and frozen variables can be considered.
- Generic Artefacts:** The generic artefacts, i.e. of a software product line, include variation points (indicated by gray shapes in Figure 3) to which one or multiple variants (indicated as colored shapes in Figure 3) can be bound. However, the way of specifying such variation points (and the related variants) depends on the used VIT. For example, using preprocessing as a VIT, the variation points might be indicated by `#if`-statements in the generic artefacts. In some situations an instantiator may also generate artefacts from scratch, which does not require any generic artefacts as an input to the instantiator.
- Variants:** The different variants that can be applied to a variation point of a generic artefact may be implemented independent from the generic artefacts. However, this also depends on the used VIT. In the example of preprocessing, the different variants will be part of the generic artefacts. The variants not selected as part of the product will then be

deleted by the preprocessor. In case of using aspect-orientation as VIT, the variants are implemented as independent aspects, which can be woven into the generic artefacts if they are selected as part of the product.

The relation between IVML configuration, generic artefacts, and variants is illustrated in Figure 4. The decision variables and their values will be passed in as a VIL configuration instance, which exactly defines the scope, while the files will be passed in as a VIL container of type FileArtifact (we will discuss this in detail in Section 3.1.3). The way of processing this information depends on the implemented instantiator logic. Figure 4 sketches two possible variants of such logic in pseudo-code:

- **Variant A:** In variant A the instantiator will process all files given by the VIL container, i.e. in terms of searching for variation points in each file (this is described as VIT-statement in Figure 4; some VITs may also introduce further concepts besides variation points that can be searched and processed by an instantiator). However, what to do if a certain variation point (or VIT-statement) is found heavily depends on the used VIT and the intention of the domain engineer who defines these variation points. Thus, we cannot give further information regarding the actual logic.
- **Variant B:** In variant B the instantiator will process all decision variables given by the VIL configuration, i.e. in terms of searching for a specific variable-value pair. However, what to do if a certain variable-value pair is found heavily depends on the used VIT and the relation between the specific decision variable and the artefacts. Thus, we cannot give further information regarding the actual logic.

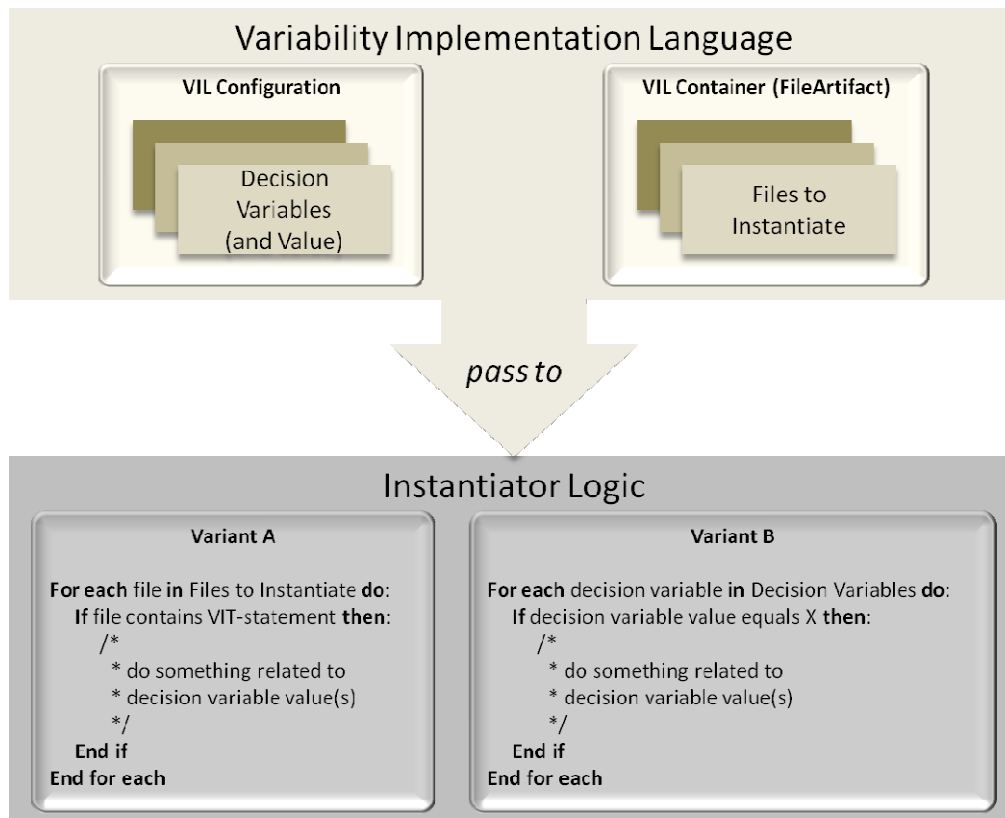


Figure 4: White-box view of a generic instantiator (simplified)

An instantiator may also provide further functionality, i.e. the generation of files based on the variable-value pairs (this may also exclude the selection of files to instantiate as the instantiation process will generate completely new files), the combination of other (non-source) artefacts like documentation, etc. However, this depends on the used VIT and the specific purpose an instantiator is designed for.

3.1.2. Eclipse Plug-in Project Creation and Configuration for New Instantiators

The first step towards a new instantiator is to create new Eclipse plug-in project: *File* → *New* → *Project...* . In the emerging wizard, open the category *Plug-in Development*, select *Plug-in Project*, and click the *Next* button. In the *New Plug-in Project* wizard, define a name for your project. We will use the following name throughout this section: *EASyExampleInstantiator*. Further, define the target platform with which the plug-in should run. In this case, the instantiator plug-in will run with a *standard OSGi framework*. Figure 5 shows how the first configuration page for the new plug-in project must look like.

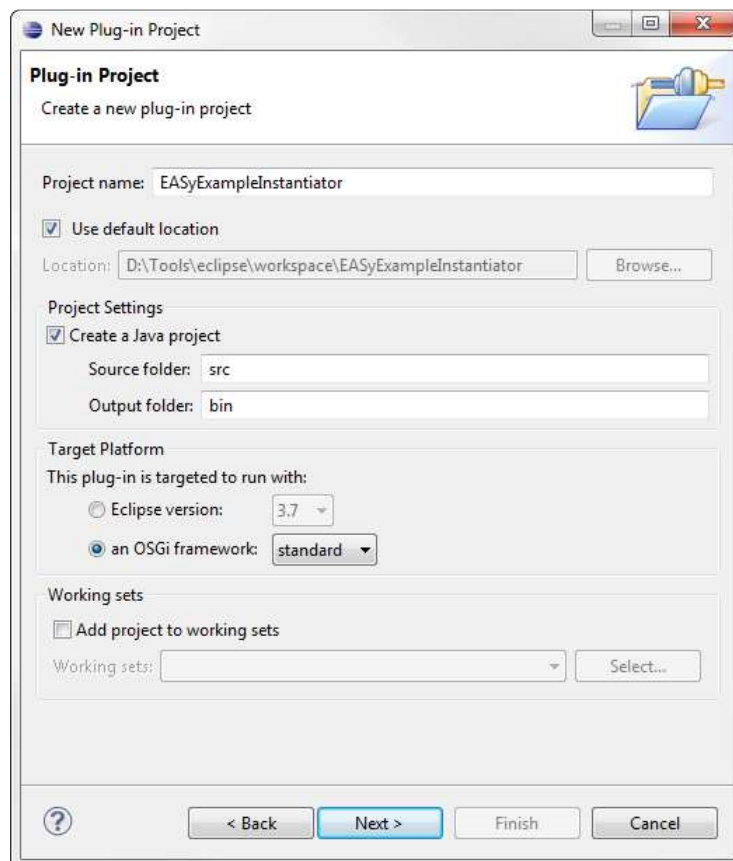


Figure 5: Configuration of a new Eclipse plug-in project for a new Instantiator

Click on the *Next* button and define the properties of your plug-in. We will use the following values for the required properties:

- *ID*: de.uni_hildesheim.sse.easy.instantiator.exampleInstantiator
- *Version*: 0.0.1
- *Name*: EASyExampleInstantiator
- *Provider*: University of Hildesheim – SSE

Leave all other properties and options as-is and finish the configuration by clicking the *Finish* button of the *New Plug-in Project* wizard. Please note that some of the following steps described in this section can also be done by using the wizard. However, we decided to do these steps manually to provide a more detailed explanation.

The plug-in manifest file will open by default. In the *Overview* tab check the *Activate this plug-in when one of its classes is loaded* checkbox and the *This plug-in is a singleton* checkbox. The first check will guarantee that the plug-in is activated when EASy-Producer loads one of its classes, while the second check is related to one of the concepts of EASy-Producer: each instantiator exists only once (only one instance) and can be accessed by any product line project. Thus, this check guarantees that the new instantiator will follow the concepts of EASy-Producer.

The next step is to define the dependencies of the new plug-in. Thus, open the plug-in manifest and select the *Dependencies* tab. On the left side click the *Add...* button in order to specify the following plug-ins:

- *org.eclipse.equinox.ds*: This plug-in simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies⁶.
- *org.eclipse.core.runtime*: This plug-in provides support for the Eclipse runtime platform, core utility methods, and the extension registry⁷. The latter is important for the EASy-Producer extension mechanism.
- *de.uni-hildesheim.sse.easy.instantiatorCore*: This plug-in provides the core capabilities of the EASy-Producer instantiator concept. We will use parts of this plug-in in Section 3.1.3.

By default, Eclipse adds the package *org.osgi.framework* as *Imported Packages* because of the selected target platform in the *New Plug-in Project* wizard. However, this package is not required for the integration with EASy-Producer and, thus, can be removed. Select the package on the right side of the *Dependencies* tab and click the *Remove* button. Then, click the *Add...* button and select *org.osgi.service.component* as *Imported Packages*. This package provides support for service components and their interaction with the context in which they are executed⁸. The *Dependencies* tab should now look like the one illustrated by Figure 6.

⁶ For more information visit: <http://eclipse.org/equinox/>

⁷ For more information visit: [Eclipse API – org.eclipse.core.runtime](#)

⁸ For more information visit: [OSGi API – org.osgi.service.component](#)

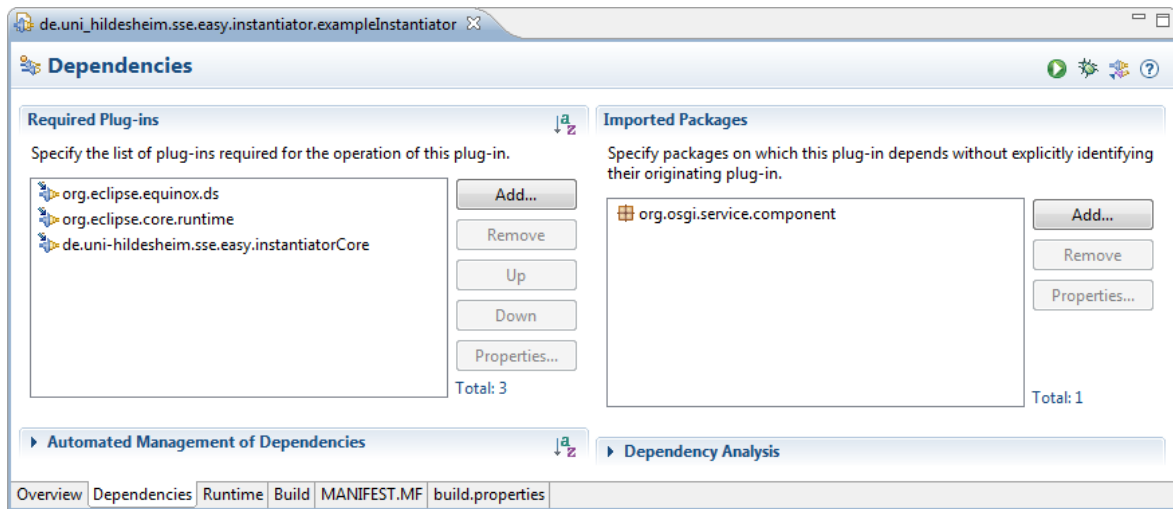


Figure 6: Definition of the required plug-ins for the new instantiator

In order to register the new plug-in to EASy-Producer, the service component has to be declared. Thus, switch to the *MANIFEST.MF* tab in the plug-in manifest and add the following *Service-Component* declaration:

```
Service-Component: OSGI-INF/instantiator.xml
```

This *Service-Component* declaration specifies the location where to find the information about the service component, which shall be integrated into EASy-Producer. The declared XML file will be defined in the next step. Figure 7 shows how the manifest file must look like.

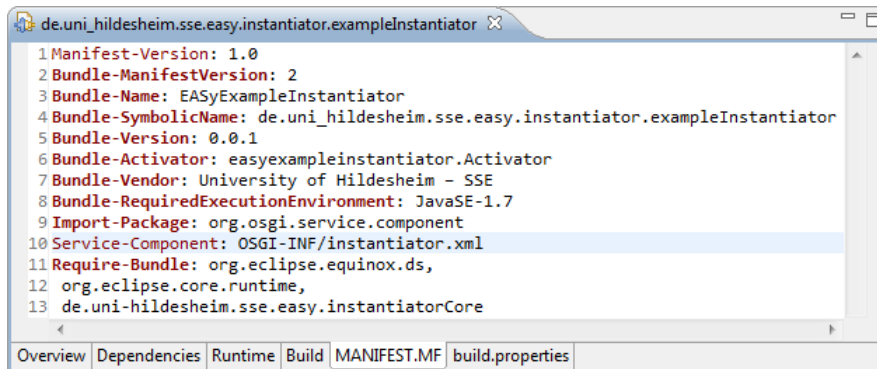


Figure 7: Declaration of the service-component for the new instantiator

The definition of the service component requires the creation of a new folder within the plug-in project. Right click on the plug-in project and select *New* → *Folder*. The name of the folder has to be *OSGI-INF*. Then, create a new XML file within this folder. Right click on the folder and select *New* → *Other...*. In the emerging wizard, open the category *XML*⁹, select *XML File*, and click the *Next* button. Define the name of the file in accordance to the file declared in the manifest illustrated in Figure 7: *instantiator.xml*. Clicking the *Finish* button will open the XML editor. Switch to the source tab and edit the file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
```

⁹ If the category *XML* does not exist, install XML support using *Help* → *Install New Software* or open the category *General*, select *File*, and define the name as well as the file-type manually.

```

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  immediate="true"
  name="EASy Example Instantiator">

  <implementation class="easyexampleinstantiator.ExampleEngine"/>

  <service>
    <provide interface="de.uni_hildesheim.sse.easy_producer.
      instantiator.InstantiatorEngine"/>
  </service>
</scr:component>

```

Figure 8 shows the final XML file. Please note that we used the names and package-structure of our example in Figure 8. Thus, with respect to different implementations the name of the service component in line 4 as well as the package and the class name of the implementation class element in line 6 (the class, which will implement the instantiator) have to be adapted. Please ignore the warning in line 6 as the class currently does not exist. This will be part of Section 3.1.3.

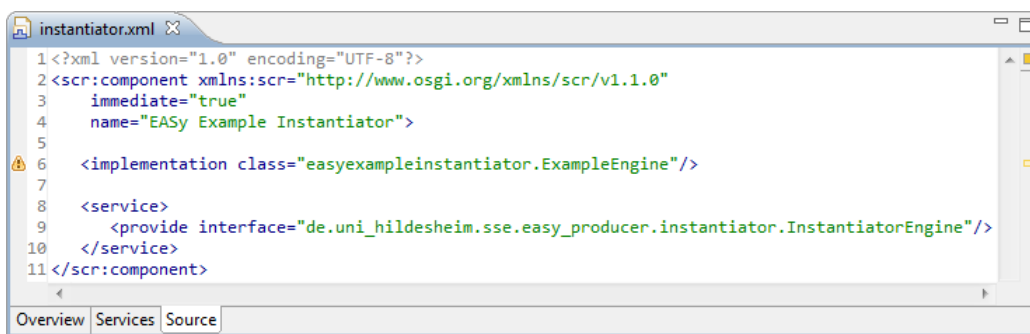


Figure 8: Definition of the service-component for the new instantiator

The previously defined XML file must be included in the binary build. Thus, open the manifest file again and switch to the *Build* tab. In the left lower part of this tab select the *OSGI-INF* folder to be included in the binary build.

The last step is the inclusion of external, third-party libraries – the actual instantiator. Please note that this step is only required if the main implementation of the instantiator or other required functionalities are implemented in another plug-in or library. In such a case, build the plug-in or the library first¹⁰. Then, right click on the current instantiator plug-in project, select *New* and *Folder*. The name of the new folder must be *lib*. Include all libraries in this folder that are required by the new instantiator. The folder and the required libraries have to be included in the *Classpath* of the new plug-in. Thus, open the plug-in manifest and switch to the *Runtime* tab. Add the libraries to the *Classpath* by clicking on the *Add...* button on the right side of the *Runtime* tab. Select all required libraries of the *lib* folder and click the *Ok* button. Switch to the *Build* tab of the plug-in manifest and select the *lib* folder to be part of the *Binary Build* in the left lower part of this tab. Figure 9 and Figure 10 show the result in the context of our example. Figure 9 shows the included library *de.uni_hildesheim.sse_0.0.1.jar*, which provides the main functionality of our prototypical instantiator and, thus, has to be available at runtime. Figure 10 illustrates the build configuration in which the library (highlighted) is selected as part of the *Binary Build*.

¹⁰ If you do not know how to build a plug-in, please consider Section 3.1.3.

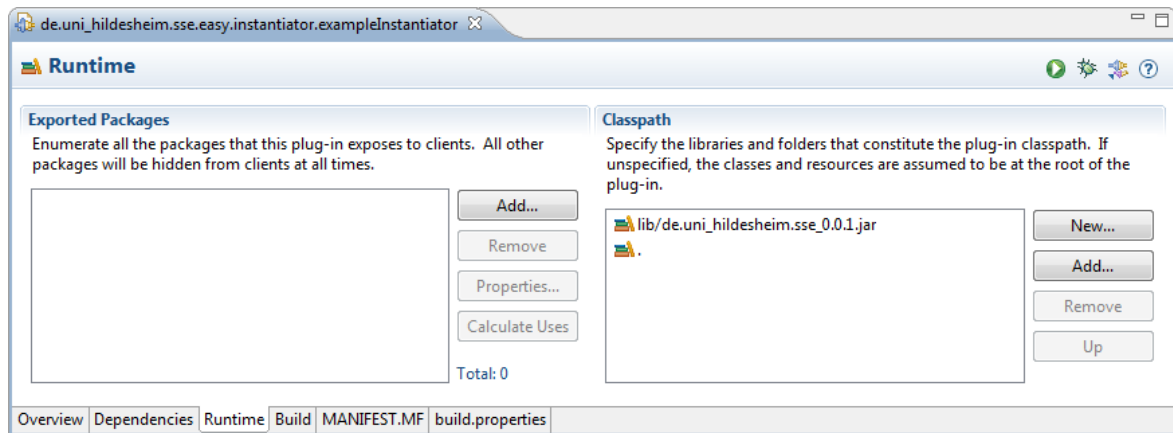


Figure 9: Classpath specification of external, required libraries

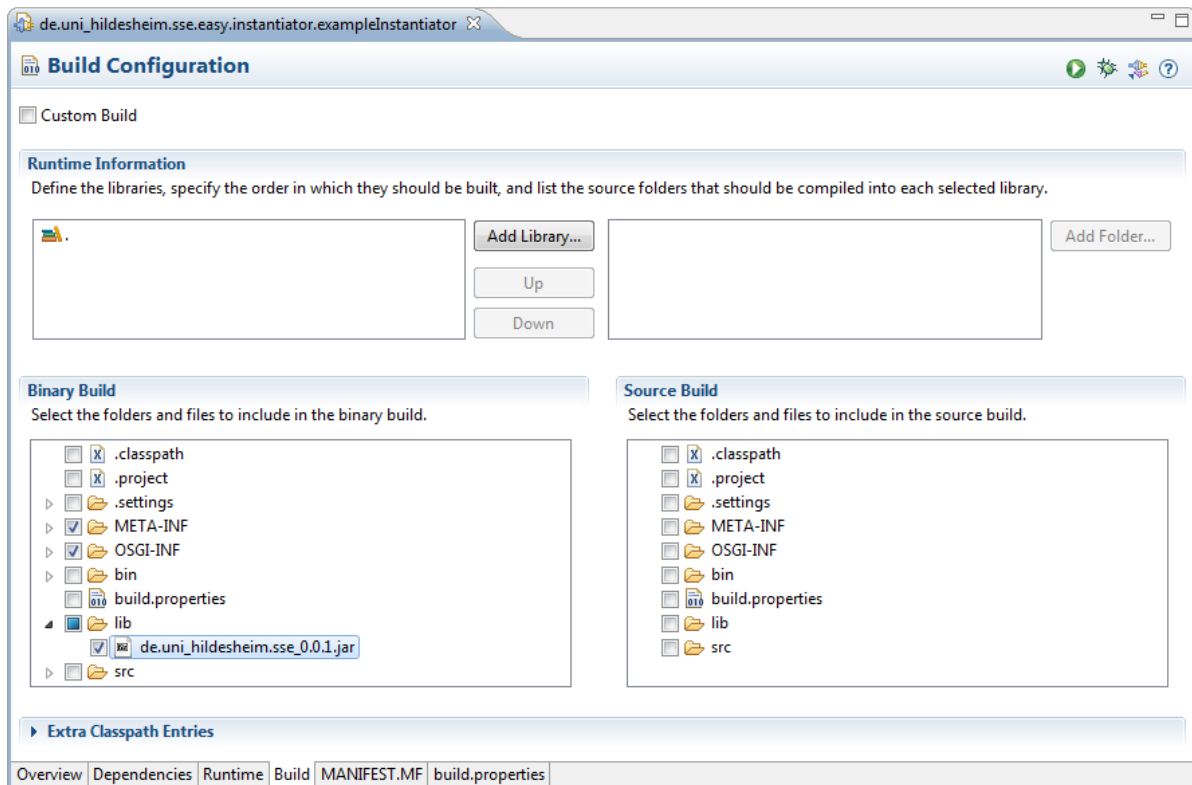


Figure 10: Binary Build selection of external, required libraries

Finally, the plug-in project is set up, configured and ready to use. In the next section, we will further develop this plug-in by implementing instantiator-specific functionality based on the results of this section.

3.1.3. Instantiator Implementation

In the previous section, we set up the Eclipse plug-in project for implementing a new instantiator for EASy-Producer. In this section, we will describe how to implement the (basic) functionalities of an instantiator. However, as each instantiator provides its individual capabilities and is used for different purposes, this description will only include the basic functionalities that are common to each instantiator.

The first step is to create a new Java class file. Right click on the package that was defined as the implementation class package in Section 3.1.2 and select *New* → *Class*. In the emerging *Java Class* wizard, define the name of the new class in accordance to the name of the implementation class (cf. Section 3.1.2). In our example, we use the name *ExampleEngine*. Leave all other options as-is.

Each instantiator must implement the *IVilType* interface, must be annotated with the annotation *Instantiator*, and must implement at least one static method, which typically has the same name as the instantiator (because the name of the method will as well identify the instantiator call in VIL). This enables the integration of the new instantiator as part of the VIL language. We will describe this intergration in detail in Section 3.1.4, while details about VIL types in general, the annotation, and, in particular, the *IVilType* interface can be obtained in the VIL language specification (cf. Section 2.4).

Thus, the next step is to edit the new class file as follows (please note that we use the packages and class names of our example):

```
package easyexampleinstantiator;

import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel
    .FileArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
    .ArtifactException;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
    .Collection;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
    .IVilType;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
    .Instantiator;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.Set;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes
    .configuration.Configuration;

@Instantiator("exampleEngine")
public class ExampleEngine implements IVilType {

    protected void activate(ComponentContext context) {
        try {
            TypeRegistry.registerType(ExampleEngine.class);
        } catch (VilException e) {
            e.printStackTrace();
        }
    }

    protected void deactivate(ComponentContext context) {
    }

    public static Set<FileArtifact> exampleEngine(Collection<FileArtifact>
        templates, Configuration config) throws ArtifactException {
        // Implementation of the actual instantiation
    }

    // ...
}
```

We will now discuss each of these methods in detail:

- **activate:** This method is used to activate the instantiator plug-in. In this case, we will register the new type in the type registry. This will include the type in the artefact model, ready for use in the VIL build language or the template language.
- **deactivate:** This method is used to deactivate the instantiator plug-in. However, in this situations we do not need to unregister the type again as this would yield errors in the VIL build script or template as the type would be unknown.

The actual implementation above is rather simple. The single static method represents the entry-point of the instantiator when it is called as part of a VIL build execution (mandatory). Here, the name of the method *exmapleEngine* will be used in the VIL language to call this instantiator, including the defined parameters. In our example, this method requires the following parameters:

- *templates:* This collection includes a set of *FileArtifacts*, which represent (real) generic artefacts, for example, of a specific software product line. The *Collection* type as well as the *FileArtifact* type are defined in the **Artifact Model** of VIL (see VIL Language Specification for details on the VIL artefact model). This set of artefacts will be processed by the instantior depending on the actual logic.
- *config:* The current configuration based on the IVML variability model of the respective product line. The *Configuration* type is again part of the VIL artefact model. The configuration provides access to the current variables and their values to determine which artefacts have to be instantiated in which way. However, this is defined in the actual implementation of the instantiator.

Please note that the *exmaple* above only provides a prototypical implementation of an instantiator. The types used in the actual implementation depend on the logic of the instantiator and its purpose. The available types in turn depend on the VIL type system.

The last step is to build the plug-in. Open the plug-in manifest file and click on the *Export deployable plug-ins and fragments* button in the upper right corner. In the emerging wizard select the current plug-in project, specify the desired destination¹³, and click on the *Finish* button.

Finally, the plug-in and, thus, the instantiator is implemented, build, and ready for use. In the next section, we will describe how to integrate a new instantiator in EASy-Producer. We will also have a quick look on how to use it. However, for detailed information on how to use an instantiator, please consider the [EASy-Producer User Guide](#).

3.1.4. Instantiator Integration

In the previous section, we implemented the (basic) functionalities of a new instantiator. Further, we build a deployable plug-in, which we will use in this section for integrating the new instantiator within an EASy-Producer installation.

The first and only step is to copy the previously build instantiator plug-in into the *dropins* folder of the Eclipse application in which EASy-Producer installed. Start the Eclipse application and create a new product line project: *File* → *New* → *Other...* → *EASy-Producer* → *New EASy-Producer Project*.

¹³ The destination is up to the developer. However, we recommend using a location, which is easy to find as we will need the location for integrating the new instantiator in Section 3.1.4.

The name of the new project is up to the developer. If a product line project is available in the workspace, open, for example, the **VIL Build Language Editor** by double clicking the VIL file in the **EASy-folder**. Calling the new instantiator as part of a build task can be done by simple typing the name of the method defined in Section 3.1.3 and passing the required parameters. Figure 12 shows the call of our example instantiator.

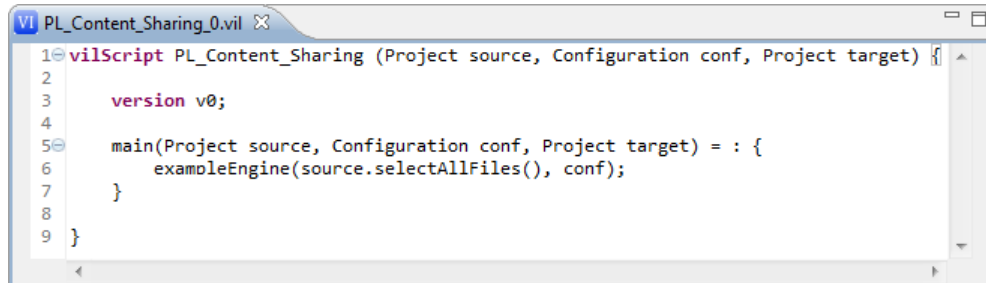


Figure 12: Using the new instantiator in a VIL build script.

3.2. Implementing a New VIL Artefact Type

The Variability Implementation Language (VIL) is a textual language for the flexible specification of the instantiation process of a software product line or any other software project that includes variabilities. Actually, VIL is not a single language. It consists of four main parts, namely the artefact model, the VIL template language, blackbox instantiators, and the VIL build language. In this section, we will focus on the artefact model and the extension of this model by new artefact types. In the first part, we will briefly introduce the VIL artefact model and discuss the basic concept regarding the extension capabilities. In the second part, we will describe the extension of the model by an example artefact in a step-wise manner.

3.2.1. The VIL Artefact Model in EASy-Producer

The artefact 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. Thus, the available artefact types will be used in the VIL build language and the VIL template language to enable the instantiation of variable artefacts of the respective type. More details on the artefact model and VIL in general can be found in the VIL language specification (cf. Section 2.4).

The classes in the artefact model can be understood as meta-classes of artefacts. Instances of these classes represent real artefacts. Artefacts are VIL types in order to be available in the VIL editors. Currently, there are five fundamental types:

- **Path expressions** for denoting file system and language-specific paths.
- **Simple artefacts**, which cannot be decomposed. Typically, generic folders and simple generic components shall be represented as simple artefacts. Some of those artefacts act as default representation through the ArtefactFactory, i.e., any real artefact which is not specified by a more specific artefact class is represented by those artefact types.

- **Fragment artefacts**, representing decomposed artefact fragments such as a Java method or a SQL statement.
- **Composite artefacts**, representing decomposable artefacts consisting of fragments. In case of resolution conflicts, composite artefacts have more priority than simple artefacts, e.g., if there is a simple artefact and a composite artefact representation of Java source classes, the composite will be taken. However, if there are resolution conflicts in the same type of artefacts, e.g., multiple composite representations, then the first one loaded by Java will take precedence. In order to implement a decomposable artefact, also an instance of the **IArtifactCreator** must be implemented. This describes creator instances which know how to translate real world objects into artifact instances. We will illustrate the usage of this interface in the implementation of our example in Section 3.2.3
- The types in `de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes` are most basic and enable building a bridge to the variability model with own VIL-specific operations.

Instances of all artefact types can be obtained from the **ArtefactFactory**. This is in particular true for instances of the **ArtefactModel** which provides an environment for instantiating artefacts, i.e., it manages created artefacts. While the **ArtefactFactory** may be used standalone, the correct internal listener registration is done by **ArtefactModel** so that model and artefacts are informed about changes and can be kept up to date, i.e., artefact instances shall be created using methods of **ArtefactModel**.

Subclassing these artefact types (and registering them with the artefact factory through the Eclipse DS mechanism) transparently leads to more specific artefact types with more specific operations. Please note that even the simple names of Vil types and artefacts shall be unique (unless they shall override existing implementations) due to the transparent embedding into the VIL languages. Types must be registered in **TypeRegistry**.

All operations marked by the annotation **Invisible** will not be available through the VIL languages. However, the (semantics of the) Invisible annotation may be inherited if required. By convention, collections are returned in terms of type-parameterized sets or sequences. However, an artefact method returning a collection must be annotated by **OperationMeta.returnGenerics()** in order to define the actual types used in the collection (this is not available via Java mechanisms). Further, operations and classes may be marked by the following annotations:

- **Conversion** to indicate type conversion operations considered for automatic type conversion when calling methods from a VIL expression. These methods must be static, take one parameter of the source type and return the target type.
- **OperationMeta** for renaming operations (for operator implementations), determining their operator type or, as mentioned above, making the type parameters of a generic return type explicit. Basically, all three information types are optional.
- **ClassMeta** for renaming the annotated class, i.e., hiding the Java implementation name.

Collections may define generic iterator operations such as checking a condition or applying a transformation expression to each element. Therefore, a non-static operation on a collection receiving (at the moment exactly) one **ExpressionEvaluator** instance as parameter (possibly more

parameters) will be considered by VIL as an iterator operation. The **ExpressionEvaluator** will carry an iterator variable of the first parameter (element type) of the collection as well as an expression parameterized over that variable (i.e., it uses the [unbound] variable). The job of the respective collection operation is to apply the expression to each element in the collection, i.e., to bind the variable to each collection element (via the runtime variable of the temporarily attached **EvaluationVisitor** in the **ExpressionEvaluator**), to call the respective evaluation operation of the **ExpressionEvaluator** and to handle the returned evaluation result appropriately.

Artefact or instantiator operations may cause VIL rules and templates to fail if they return a non-true result, i.e., an empty collection or null. However, in order to state explicitly that an operation cannot be executed, an operation shall throw an **ArtefactException**.

Basically, artefact or instantiator operations are identified by their name, the number, sequence and type of their parameter. However, some operations such as template processors may require an unlimited number of not previously defined parameters. In this case, VIL allows to pass in named parameters. In the respective artefact or instantiator operations, named parameters are represented by a **Map** as last parameter which receives the names and the actual values of given named VIL parameters. The interpretation of named parameters belongs to the respective method.

3.2.2. Eclipse Plug-in Project Creation and Configuration for New Artefacts

The set up of an Eclipse plug-in project for a new VIL artefact type is quite similar to the set up for a new instantiation described in Section 3.1.2. Below, we will describe the only changes with respect to the instantiator set up:

- **Project name:** We will use `EASyExampleArtifact` as the name for the project throughout this sections.
- **OSGI information:** The XML-file for the definition of the service component will change in terms of the name (`artifact.xml`) and the content as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
    immediate="true"
    name="EASy Example Artifact">

    <implementation class="easyexampleartifact.ExampleArtifact"/>

    <service>
        <provide interface="de.uni_hildesheim.sse.easy_producer.
            instantiator.model.vilTypes.IVilType"/>
    </service>
</scr:component>
```

3.2.3. Artefact Implementation

The first step is to create a new Java class file. In our example, we use the name *ExampleArtifact*. Each new artefact has to implement the *IVilType* interface and may extend one of the base VIL

types introduced in Section 3.2.1. We will extend the **FileArtifact** in this example. Thus, the content¹⁴ of the new class file looks like this:

```
package easyexampleartifact;

import java.io.File;

import org.osgi.service.component.ComponentContext;

import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.ArtifactCreator;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.ArtifactModel;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.FileArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.FragmentArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.IArtifactVisitor;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.representation.
    Binary;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.representation.Text;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.ArtifactException;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.IVilType;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.Set;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.TypeRegistry;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.VilException;

@ArtifactCreator(ExampleFileArtifactCreator.class)
public class ExampleArtifact extends FileArtifact implements IVilType {

    public ExampleArtifact(File file, ArtifactModel model) {
        super(file, model);
    }

    protected void activate(ComponentContext context) {
        try {
            TypeRegistry.registerType(ExampleArtifact.class);
        } catch (VilException e) {
            e.printStackTrace();
        }
    }

    protected void deactivate(ComponentContext context) {
    }

    @Override
    public void delete() throws ArtifactException {
        // Here goes the implementation
    }

    @Override
    public String getName() throws ArtifactException {
        // Here goes the implementation
    }

    @Override
    public void rename(String name) throws ArtifactException {
        // Here goes the implementation
    }
}
```

¹⁴ Please note that we cannot discuss all methods in detail due to the complexity of the artefact model and the available types and methods.

```
@Override
public void accept(IArtifactVisitor visitor) {
    // Here goes the implementation
}

@Override
public boolean isUptodate(long timestamp) {
    // Here goes the implementation
}

@Override
public boolean exists() {
    // Here goes the implementation
}

@Override
public void artifactChanged() throws ArtifactException {
    // Here goes the implementation
}

@Override
public Set<? extends FragmentArtifact> selectAll() {
    // Here goes the implementation
}

@Override
protected Text createText() throws ArtifactException {
    // Here goes the implementation
}

@Override
protected Binary createBinary() throws ArtifactException {
    // Here goes the implementation
}

@Override
public void store() throws ArtifactException {
    // Here goes the implementation
}
}
```

We will now discuss each of these methods in detail:

- **Constructor:** The constructor of this class requires a (real) artefact in terms of a file, which will be represented by this artefact type, and the corresponding artefact model instance this artefact belongs to. These parameters are passed to the super-class, the **FileArtifact**.
- **activate:** This method is used to activate the instantiator plug-in. In this case, we will register the new type in the type registry. This will include the type in the artefact model, ready for use in the VIL build language or the template language.
- **deactivate:** This method is used to deactivate the instantiator plug-in. However, in this situations we do not need to unregister the type again as this would yield errors in the VIL build script or template as the type would be unknown.
- **delete:** This method deletes the current instance of this artefact Including its underlying real-world object, e.g., this operation may delete an entire file.
- **getName:** This method returns the name of the current instance of this artefact.

- **rename:** This method renames the current instance of this artefact and its underlying real-world object.
- **accept:** This method visits the current instance of this artefact (and dependent on the visitor also contained artifacts and fragments) using the given visitor.
- **isUptodate:** This method returns whether the current instance of this artefact is up-to-date (with respect to the given timestamp) and whether it shall be considered for recreation in preconditions of VIL build language rules.
- **exists:** This method returns whether the current instance of this artefact exists. Also this method is considered by the VIL build language.
- **artifactChanged:** This method is called when the current instance of this artefact was changed, e.g., to trigger a reanalysis of substructures. This may be caused by one of the alternative basic representations such as text or binary (see below).
- **selectAll:** This method returns all artefacts of the current instance of this composite artefact is composed of.
- **createText:** This method actually creates a text representation of the current instance of this artefact. The binary representation enables to modify the entire artifact from a textual point of view, i.e., using text manipulation operations. Please note that the **getText**-method of the super-class **CompositeArtifact** calls this method and registers the listeners appropriately.
- **createBinary:** This method actually creates a binary representation of the current instance of this artefact. The binary representation enables to modify the entire artifact from a binary point of view, i.e., in terms of individual bytes. Please note that the **getBinary()**-method of the super-class **CompositeArtifact** calls this method and registers the listeners appropriately.
- **store:** This method stores changes to the artefact. Typically, this is done by saving the changes of the file contents to the real-world file artefact.

The new artefact type is derived from the **FileArtifact** type of the VIL artefact model, which is derived from the **CompositeArtifact** introduced in Section 3.2.1. This type of artefact also requires the implementation of an instance of the **IArtifactCreator** interface to relate real-world artefacts to the new VIL type (indicated by the annotation of this class-implementation above). Implementations of this artefact must fulfill the following contract:

- The method **handlesArtifact(Class, Object)** of the **ArtifactCreator** is called to figure out whether a creator (**Class**) is able to handle a certain artifact (**Object**) under given class-based restrictions. Typically, more specific creators are asked later than more generic ones, but more specific creators (according to inheritance relationships) are considered first for creation. An implementation which answers `<code>true</code>` must be able to create the queried artifact.
- The method **createArtifactInstance(Object)** of the **ArtifactCreator** actually creates an instance for the previously queried object. However, no information shall be stored nor there is a guarantee that this method will be called (dependent on the other registered creators). As stated above, if **handlesArtifact(Class, Object)** answers with `true`, **createArtifactInstance(Object)** must be able to perform the creation for the given object.

In our exmaple, we will define a new class for the implementation of an artefact creator, named `ExampleFileArtifactCreator`. The implementation of this class is given below:

```
package easyexampleartifact;

import java.io.File;

import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.ArtifactModel;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.
    DefaultFileArtifactCreator;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.FileArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.artifactModel.IArtifact;
import de.uni_hildesheim.sse.easy_producer.instantiator.model.vilTypes.ArtifactException;

public class ExampleFileArtifactCreator extends DefaultFileArtifactCreator {

    @Override
    protected boolean handlesFileImpl(File file) {
        return checkSuffix(file, ".example");
    }

    @Override
    public FileArtifact createArtifactInstance(Object real, ArtifactModel model)
        throws ArtifactException {
        return new ExampleArtifact((File) real, model);
    }

    @Override
    public Class<? extends IArtifact> getArtifactClass() {
        return ExampleArtifact.class;
    }
}
```

We will now discuss each of these methods in detail:

- **handlesFileImpl:** This method may specify additional properties of a file that must be fulfilled in order to define the file as representable by this artefact type (the **ExampleArtifact** in this case). It is already guaranteed that the passed file is a file and not a directory. In our exmaple, we will check whether the suffix of the given file matches `“.example”`.
- **createArtifactInstance:** This method creates a new instance of the artefact type based on the passed (real) artefact.
- **getArtifactClass:** This method return the class that implements the artefact.

The last step is to build the plug-in. Open the plug-in manifest file and click on the *Export deployable plug-ins and fragments* button in the upper right corner. In the emerging wizard select the current plug-in project, specify the desired destination¹⁵, and click on the *Finish* button.

Finally, the plug-in and, thus, the new artefact is implemented, build, and ready for use. The integration is the same as in the instantiator case described in Section 3.1.4 (copy the final plug-in into the dropins-folder). The next time Eclipse will be started the new artefact type can be used in the VIL build language and in the VIL template language.

¹⁵ The destination is up to the developer. However, we recommend using a location, which is easy to find as we will need the location for integrating the new instantiator in Section 3.1.4.

3.3. Implementing a New Reasoner

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. Similar to the instantiators in EASy-Producer, we provide a simple extension mechanism for integrating custom reasoners with the tool.

In the following sections, we will describe the set-up a new plug-in project in Eclipse for implementing a custom reasoner. This also includes the specific configurations that have to be done to utilize the automated search and integration mechanism provided by EASy-Producer. Further, we will discuss the methods that are required when implementing a new reasoner.

3.3.1. Eclipse Plug-in Project Creation and Configuration for New Reasoners

The first steps of the creation of a new Eclipse plug-in for the implementation of a new reasoner are quite similar to the creation of a new instantiator plug-in (cf. Section 3.1.2). However, the first change is in the name of the new project. We will use *EASyExampleReasoner* throughout this section as the name and define, again, the *standard OSGI framework* as the target platform.

Further changes occur in the definition of the plug-in properties. We will use the following values:

- *ID*: `de.uni_hildesheim.sse.easy.reasoner.exampleReasoner`
- *Version*: `0.0.1`
- *Name*: `EASyExampleReasoner`
- *Provider*: `University of Hildesheim – SSE`

The plug-in manifest file will open by default after clicking the *Finish* button. In the *Overview* tab check the *Activate this plug-in when one of its classes is loaded* checkbox and the *This plug-in is a singleton* checkbox.

The next step is to define the dependencies of the new plug-in. Thus, open the plug-in manifest and select the *Dependencies* tab. On the left side click the *Add...* button in order to specify the following plug-ins:

- `org.eclipse.equinox.ds` (described in Section 3.1.2)
- `org.eclipse.core.runtime` (described in Section 3.1.2)
- *ReasonerCore*: This plug-in provides the core capabilities of the EASy-Producer reasoning concept. We will use parts of this plug-in in Section 3.3.2.
- `de.uni_hildesheim.sse.varModel`: This plug-in provides access to the underlying variability object model of EASy-Producer. This provides, for example, the access to the current configuration of the variability model, the included decision variables and constraints, etc. We will use parts of this plug-in in Section 3.3.2.

By default, Eclipse adds the package `org.osgi.framework` as *Imported Packages* because of the selected target platform in the *New Plug-in Project* wizard. However, this package is not required for the integration with EASy-Producer and, thus, can be removed. Select the package on the

right side of the *Dependencies* tab and click the *Remove* button. Then, click the *Add...* button and select *org.osgi.service.component* as *Imported Packages*. This package provides support for service components and their interaction with the context in which they are executed.

In order to register the new plug-in to EASy-Producer, the service component has to be declared. Thus, switch to the *MANIFEST.MF* tab in the plug-in manifest and add the following *Service-Component* declaration:

```
Service-Component: OSGI-INF/reasoner.xml
```

The definition of the service component requires the creation of a new folder within the plug-in project. Right click on the plug-in project and select *New → Folder*. The name of the folder has to be *OSGI-INF*. Then, create a new XML file within this folder. Right click on the folder and select *New → Other...*. In the emerging wizard, open the category XML¹⁶, select *XML File*, and click the *Next* button. Define the name of the file in accordance to the file declared in the manifest: *reasoner.xml*. Clicking the *Finish* button will open the XML editor. Switch to the source tab and edit the file as follows:

```
<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
  immediate="true"
  name="EASy Example Reasoner">

  <implementation class="easyexamplereasoner.ExampleReasoner"/>

  <service>
    <provide interface="de.uni_hildesheim.sse.reasoning.core.
      reasoner.IReasoner"/>
  </service>
</scr:component>
```

The previously defined XML file must be included in the binary build. Thus, open the manifest file again and switch to the *Build* tab. In the left lower part of this tab select the *OSGI-INF* folder to be included in the binary build.

In order to include external, third-party libraries, follow the last step described in Section 3.1.2.

Finally, the plug-in project is set up, configured and ready to use. In the next section, we will further develop this plug-in by implementing the required classes and methods based on the results of this section.

3.3.2. Reasoner Implementation

In the previous section, we set up the Eclipse plug-in project for implementing a new reasoner for EASy-Producer. In this section, we will first describe the different classes that are required to register the new reasoner in EASy-Producer and, second, describe how to implement the required (basic) methods of a reasoner. However, as each reasoner provides its individual capabilities, this description will only include the basic functionalities that are common to each reasoner.

¹⁶ If the category XML does not exist, install XML support using *Help → Install New Software* or open the category *General*, select *File*, and define the name as well as the file-type manually.

In contrast to the instantiator implementation, the implementation of a reasoner requires a two Java classes. Below, we will describe each required class in detail. Please note that the names of the classes are due to the name of this example.

- *ExampleReasonerDescriptor.java*: This class includes the following attributes of a reasoner: the descriptive name, the version, the license, license restrictions, and a download source (if this is a third-party reasoner). While the name is a mandatory attribute, all other attributes are optional. In this example, the reasoner descriptor looks like this:

```
package easyexamplereasoner;

import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasonerDescriptor;

public class ExampleReasonerDescriptor extends ReasonerDescriptor {

    static final String NAME = "Example Reasoner";

    static final String VERSION = "0.1";

    private static final String LICENSE = "<Licences Agreement>";

    public ExampleReasonerDescriptor() {
        super(NAME, VERSION, LICENSE, null, null);
    }

    @Override
    public boolean isReadyForUse() {
        return true;
    }
}
```

- *ExampleReasoner.java*: This class provides the actual implementation of the reasoner. In this example, the reasoner implementation looks like this:

```
package easyexamplereasoner;

import java.net.URI;
import java.util.List;

import org.osgi.service.component.ComponentContext;

import de.uni_hildesheim.sse.model.progress.ProgressObserver;
import de.uni_hildesheim.sse.model.varModel.Constraint;
import de.uni_hildesheim.sse.model.varModel.Project;
import de.uni_hildesheim.sse.reasoning.core.frontend.ReasonerFrontend;
import de.uni_hildesheim.sse.reasoning.core.reasoner.EvaluationResult;
import de.uni_hildesheim.sse.reasoning.core.reasoner.IReasoner;
import de.uni_hildesheim.sse.reasoning.core.reasoner.IReasonerMessage;
import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasonerConfiguration;
import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasonerDescriptor;
import de.uni_hildesheim.sse.reasoning.core.reasoner.ReasoningResult;

public class ExampleReasoner implements IReasoner {

    private static final ReasonerDescriptor DESCRIPTOR = new
ExampleReasonerDescriptor();
```



```
protected void activate(ComponentContext context) {
    ReasonerFrontend.getInstance().getRegistry().register(this);
}

protected void deactivate(ComponentContext context) {
    ReasonerFrontend.getInstance().getRegistry().unregister(this);
}

@Override
public ReasonerDescriptor getDescriptor() {
    return DESCRIPTOR;
}

@Override
public ReasoningResult upgrade(URI url, ProgressObserver observer) {
    // Here goes the implementation
}

@Override
public ReasoningResult isConsistent(Project project,
    ReasonerConfiguration reasonerConfiguration,
    ProgressObserver observer) {
    // Here goes the implementation
}

@Override
public void notify(IReasonerMessage message) {
    // Here goes the implementation
}

@Override
public ReasoningResult check(Project project,
    de.uni_hildesheim.sse.model.confModel.Configuration cfg,
    ReasonerConfiguration reasonerConfiguration,
    ProgressObserver observer) {
    // Here goes the implementation
}

@Override
public ReasoningResult propagate(Project project,
    de.uni_hildesheim.sse.model.confModel.Configuration cfg,
    ReasonerConfiguration reasonerConfiguration,
    ProgressObserver observer) {
    // Here goes the implementation
}

@Override
public EvaluationResult evaluate(Project project,
    de.uni_hildesheim.sse.model.confModel.Configuration cfg,
    List<Constraint> constraints,
    ReasonerConfiguration reasonerConfiguration,
    ProgressObserver observer) {
    // Here goes the implementation
}
}
```

We will now discuss each of these methods in detail:

- **activate:** This method is used to activate the reasoner plug-in. We recommend not changing this method in order to guarantee that EASy-Producer activates the reasoner properly.
- **deactivate:** This method is used to deactivate the reasoner plug-in. We recommend not changing this method in order to guarantee that EASy-Producer deactivates the reasoner properly.
- **getDescriptor:** This method returns the descriptor of this reasoner define above stating common information about this reasoner.
- **upgrade:** This method updates the installation of this reasoner, e.g., in order to obtain a licensed reasoner version if a third-party reasoner is used.
- **isConsistent:** This method is invoked by EASy-Producer if a given variability model should be checked for satisfiability. However, the actual implementation of this method depends on the reasoner.
- **notify:** This method is called when a reasoner message is issued.
- **check:** This method checks the configuration according to the given project structure.
- **propagate:** This method checks the configuration according to the given model and propagates values, if possible. The concept of value propagation defines the automatic assignment of currently unassigned decision variables of the configuration. This automation requires the assignment of a subset of the available decision variables and the relation of these variables to the unassigned variables in terms of constraints in the variability model.
- **evaluate:** This method evaluates a given list of constraints (in the sense of boolean conditions) which are related to and valid in the context of the given project and configuration.

The last step is to build the plug-in. Open the plug-in manifest file and click on the *Export deployable plug-ins and fragments* button in the upper right corner. In the emerging wizard select the current plug-in project, specify the desired destination¹⁷, and click on the *Finish* button.

Finally, the plug-in and, thus, the reasoner is implemented, build, and ready for use. In the next section, we will describe how to integrate a new reasoner in EASy-Producer.

3.3.3. Reasoner Integration

In the previous section, we implemented the (basic) functionalities of a new reasoner. Further, we build a deployable plug-in, which we will use in this section for integrating the new reasoner in an EASy-Producer installation.

The first and only step is to copy the previously build reasoner plug-in into the *dropins* folder of the Eclipse application in which EASy-Producer installed. Start the Eclipse application and open the preference page of EASy-Producer: *Window* → *Preferences* → *EASy-Producer*. Expand the EASy-Producer category and select *Reasoners*. The new reasoner will be available as shown in Figure 13.

¹⁷ The destination is up to the developer. However, we recommend using a location, which is easy to find as we will need the location for integrating the new reasoner in Section 3.3.3.

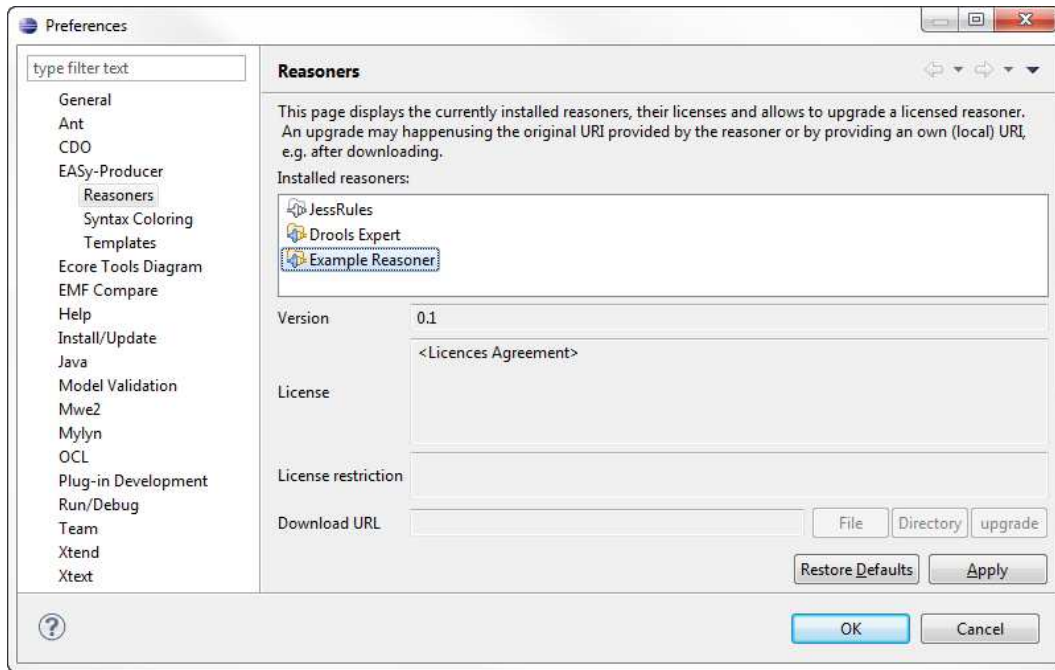


Figure 13: Reasoner preferences page

INDENICA Variability Modeling Language: Language Specification

Version 1.18

(corresponds to IVML bundle version 0.6.0)

Software Systems Engineering (SSE)

University of Hildesheim

31141 Hildesheim

Germany

Abstract

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

In this document we specify the concepts of the INDENICA variability modelling language (IVML) to describe customization and configuration options in service (platform) ecosystems.

Version

1.0	15. February 2012	first version derived from D2.1
1.01	29. February 2012	“v” as prefix in version number (technical reasons)
1.02	15. June 2012	all parenthesis follow after all “with” keywords, accessing enum literals, camelcase for refTo and refBy
1.03	17. July 2012	DSL syntax corrected
1.04	19. July 2012	Constraint syntax, operation signatures and semantics, grammar section (prepared), technical section, import conventions, clarifications in using constraints
1.05	22. July 2012	grammar revised and added to document
1.06	06. August 2012	typeSelect, typeReject, side effects, undefined values
1.07	10. August 2012	examples testcased, enum access corrected in examples, container type definition adjusted (syntax overlap with variable declaration)
1.08	16. August 2012	Details for identifiers added, technical section deleted (see IVML user guide)
1.09	28. November 2012	Operator precedence in grammar corrected, constraint type added
1.10	12. December 2012	Mass assignment of attribute values
1.11	8. January 2013	Assignment operator clarifications (‘=’ vs. ‘==’)
1.12	17. January 2013	Eval clarified
1.13	11. February 2013	Compound / container initializers and attributes clarified
1.14	12. February 2013	Grammar cleanup, decision variable naming corrected
1.15	14. February 2013	Further clarifications on ‘=’
1.16	10. July 2013	Qualification clarification (reasoner dependent)
1.17	16. August 2013	Clarification on attributes, null values introduced

Document Properties

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

Table of Contents

Table of Contents	3
Table of Figures	6
1 Introduction	7
2 The INDENICA Variability Modelling Approach	8
2.1 INDENICA Variability Modelling Core Language	9
2.1.1 Projects	9
2.1.2 Types	10
2.1.2.1 Basic Types	10
2.1.2.2 Enumerations	11
2.1.2.3 Container Types	11
2.1.2.4 Type Derivation and Restriction	12
2.1.2.5 Compounds	13
2.1.3 Decision Variables	14
2.1.4 Constraints	16
2.1.5 Configurations	24
2.2 Advanced Concepts of the INDENICA Variability Modelling Language	25
2.2.1 Attributes	25
2.2.2 Advanced Compound Modelling	28
2.2.2.1 Extending Compounds	28
2.2.2.2 Referencing Elements	29
2.2.3 Advanced Project Modelling	31
2.2.3.1 Project Versioning	31
2.2.3.2 Project Composition	32
2.2.3.3 Project Interfaces	34
2.2.4 Advanced Configuration	36
2.2.4.1 Partial Configurations	37
2.2.4.2 Freezing Configurations	38
2.2.4.3 Partial Evaluation	40
2.2.5 Including DSLs	42
3 Constraints in IVML	44
3.1 IVML constraint language	44

3.1.1	Keywords.....	44
3.1.2	Prefix operators	44
3.1.3	Infix operators.....	44
3.1.4	Precedence rules.....	45
3.1.5	Datatypes	45
3.1.6	Type conformance	46
3.1.7	Type operations	46
3.1.8	Side effects.....	47
3.1.9	Undefined values	47
3.1.10	If-then-else-endif Expressions	47
3.1.11	Let Expressions.....	47
3.1.12	User-defined operations.....	47
3.1.13	Collection operations.....	48
3.2	Built-in operations	50
3.3	Internal Types	51
3.3.1	AnyType	51
3.3.2	MetaType	51
3.4	Basic Types.....	51
3.4.1	Real.....	51
3.4.2	Integer.....	52
3.4.3	Boolean	53
3.4.4	String.....	53
3.5	Enumeration Types	54
3.5.1	Enum	54
3.5.2	OrderedEnum.....	54
3.5.3	Constraint.....	55
3.6	Collection Types	55
3.6.1	Collection	55
3.6.2	Set	56
3.6.3	Sequence.....	57
3.7	Compound Types	58
4	IVML Grammar.....	59
4.1	Basic modeling concepts.....	59

4.2	Basic types and values	61
4.3	Advanced modeling concepts	62
4.4	Basic constraints	63
4.5	Advanced constraints.....	67
4.6	Terminals.....	67
References		69

Table of Figures

Figure 1: IVML type hierarchy.....20

Figure 2: IVML type hierarchy.....46

1 Introduction

This document specifies the INDENICA variability modelling language (IVML) in terms of a live document containing the current version based on discussions with the partners and experiences made during the project.

2 The INDENICA Variability Modelling Approach

In this section, we will describe the concepts of the INDENICA Variability Modelling Language (IVML). In accordance to the previous sections, we will distinguish between a core modelling language and an advanced modelling language that extends the core language to satisfy the specific requirements that arise in the INDENICA project. This distinction facilitates ease of use for the most standard issues in variability modelling as it does not complicate the use of this language for users who do not need the more advanced features. The concepts of the core modelling language are based on the results of the discussion in D2.1. In this section, we discussed different levels of expressiveness for basic variability modelling in INDENICA. The core modelling language is extended by advanced modelling concepts that we identified as prerequisites to effective and efficient variability modelling in service-based systems and, in particular, in service (platform) ecosystems in D2.1.

The basic concepts of the IVML are related to approaches like the Text-based Variability Language (TVL) [2], the Class Feature Relationships (Clafer) [1], the Compositional Variability Management framework (CVM) [7], etc. However, we decided to develop a different approach, based on decision modelling concepts, in order to appropriately address the requirements identified in D2.1.

We will introduce a textual specification to describe the IVML concepts. This will help to give a precise representation of the modelling concepts. The syntax, we use in this section was developed as a basis for representing the concepts. Our presentation of the IVML-syntax draws upon typical concepts used in programming languages, in particular Java, and other modelling languages such as TVL [2], Clafer [1], the Object Constraint Language (OCL) [4], or the UML [5]. The dependency management concepts of the IVML mostly rely on the concepts of the OCL. We will adapt these concepts as needed to provide additional operations required by IVML-specific modelling elements, e.g. match and substitute operations for decision variables of type string.

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

- The syntax as well as the examples will be illustrated in `Courier New`.
- **Keywords** will be highlighted using bold font.
- *Elements and expressions* that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number. We recommend that the identifiers of new types start with a capital letter to easily distinguish them from variables.
- Expressions will be separated using semicolon “;”.

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

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

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

In Section 2.1, we will describe the INDENICA variability modelling core language. We will introduce the required elements and expressions to define a basic configuration space including Boolean and non-Boolean variabilities. We will further describe the dependency management capabilities of this language to restrict configuration spaces. Finally, we will describe the definition of (product) configurations based on configuration spaces.

In Section 2.2 we will describe the advanced concepts of the INDENICA variability modelling language. We will introduce extensions that are required to satisfy the specific requirements in the INDENICA project like the support for service-ecosystems, for service technology and meta-variability.

2.1 INDENICA Variability Modelling Core Language

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

2.1.1 Projects

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

Syntax:

```
project name {
```

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

}
```

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

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

Example:

```
project contentSharing {

  /* This will define a new project for a content-sharing
  project. This is related to our running example in D2.1
  */

}
```

2.1.2 Types

In a project (cf. Section 2.1.1) different kinds of core modelling elements may be used to both represent the variabilities and define a configuration space appropriately. We will express these kinds as formal types in IVML, thus defining a (strongly) typed language. We distinguish between basic types, enumerations, container types, derived and restricted types and compound types. These types can be used to declare or define concrete decision variables. Basically, all decision variables can be unset using the null keyword, i.e., explicitly assigning no value to a variable.

2.1.2.1 Basic Types

In D2.1, we argued that non-Boolean variability is a must for the core expressiveness of the INDENICA language. Thus, the IVML supports as basic types Boolean (**Boolean**), integer (**Integer**), real (**Real**) and string (**String**) with their usual meaning. The names of the basic types are aligned to OCL [4]. These types support the definition of basic variabilities, e.g. the **Boolean** type may be used for modelling optional variabilities. In addition, types like **Integer** or **Real** provide a basis for defining advanced variabilities, e.g. using an **Integer** to define a quantitative property for Quality of Service (QoS) as described in D2.1. In addition, IVML provides the basic type **Constraint** which allows declaring constraints themselves as variable.

2.1.2.2 Enumerations

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

Syntax:

```
enum Name1 {value1, ..., valuen};  
enum Name2 {value1=n1, ..., valuen=nn};
```

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

- The keyword **enum** defines that the identifier *Name* is defined as a new enumeration.
- *Name* is an identifier and defines the name of the new type.
- The identifiers surrounded by curly brackets are the concrete elements of the enumeration. A specific element of an enumeration can be accessed using the “.”-notation, e.g. *Name₁.value₁*.
- Specifying concrete numeric values for elements of an enumeration (*value_i=n_i*) turns the enumeration into an ordered enumeration. This enables relations like greater than (>) or less than (<) and operations like next (**next**) or previous (**previous**) on the values to be used.

Example:

```
enum Colors {green, yellow, black, white};  
enum BindingTimes {configuration=0, compile=1,  
runtime=2};
```

2.1.2.3 Container Types

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

The IVML supports a set of operations specific for container types, e.g. adding or appending elements to a container, deleting elements of a container, selecting specific elements, etc. We will introduce the full set of operations in Section 2.1.4.

Syntax:

```
// Declaration of a new sequence and a new set.  
sequenceOf(Type) variableName1;
```

```
setOf(Type) variableName2;
```

```
/* Access to elements of a sequence. Sets do not have  
index-based access. We will discuss variables in Section  
2.1.3. */
```

```
variableName1[index] = value;
```

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

- The **sequenceOf** and **setOf** keywords refer to a container of the respective type followed by the *Type* of the elements contained in brackets.
- The identifiers *Name*₁ and *Name*₂ are the names of the new containers.
- Accessing a specific element of a sequence container type (variable) requires the specification of an index ([*index*]). An index is either “0” or a positive integer value specifying the position of an element in a container. Accessing a specific position is only a valid operation, if this position has previously been set by different means like the **add** function (the set of operations is introduced in Section 2.1.4).

Example:

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

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

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

```
sequenceOf(ContentType) basicContents =  
  {ContentType.text, ContentType.audio};
```

2.1.2.4 Type Derivation and Restriction

The IVML allows the derivation of new types based on existing types. This supports extensibility and adaptability as users may define their own types based on basic types, enumerations or container types as well as on previously derived types. The derivation may also include restrictions to the existing type, e.g. to restrict the possible values of the new type to a subset of the values of the existing type. The restrictions are defined by one or more constraints (we will discuss constraints in detail below). Multiple constraints are implicitly combined by a Boolean OR. Thus, at least one constraint has to be satisfied by the new type. The constraints will be defined in OCL style as described in Section 2.1.4.

Syntax:

```
typedef Name1 Type ;  
  
typedef Name2 Type with (constraint1, ..., constraintn) ;
```

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

- The **typedef** keyword indicates the derivation of a new type based on an existing type.
- The identifiers *Name*₁ and *Name*₂ are the names of the new types.
- The identifier *Type* denotes the basic type from which the new type (*Name*₁ or *Name*₂) will be derived.
- The optional keyword **with** introduces a non-empty set of constraints (c.f. Section 2.1.4), surrounded by brackets, out of which at least one must hold for *Name*₂. In case of deriving *Name*₂ from **String** the constraints may define regular expressions.

Example:

```
/* Definition of a type "AllowedBitrates" which is a set  
of Integers, i.e. a kind of alias for a complex type  
definition. */  
  
typedef AllowedBitrates setOf(Integer) ;  
  
  
/* A new modelling type of the basic type integer that is  
restricted to assume values between "128" and "256". */  
  
typedef Bitrate Integer with (Bitrate >= 128 and  
Bitrate <= 256) ;
```

2.1.2.5 Compounds

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

Syntax:

```
compound Name {  
    Type name1 ;  
    ...  
}
```


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

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

Example:

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

```
compound Content {

    String name;

    Integer bitrate;

}
```

2.1.3 Decision Variables

The types introduced in Section 2.1.2 can be used to declare (decision) variables representing a concrete variability. A decision variable is an element of a project (configuration space) that basically accepts any value of its type. Constraints may further restrict the possible values by removing certain combinations of values from the allowed configuration space. The value given to a decision variable defines the variant of the represented variability.

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

Syntax:

```
// Declaration of a decision variable.
```

```
Type name1;
```

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

```
Type name2 = valueAssignment;
```

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

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

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

Example:

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

```
Integer bitrate = 128;
```

```
/* Declaration of a new variable of type enumeration with
a default value (cf. Section 2.1.2.2). */
```

```
Colors backgroundColor = Colors.black;
```

```
/* Declaration of a new variable of type container
(sequence) with default values (cf. Section 2.1.2.3). */
```

```
sequenceOf(ContentType) baseContent =  
    {ContentType.text, ContentType.audio};  
  
/* Declaration of a new variable of type compound with  
default values (cf. Section 2.1.2.5). */  
  
Content complexContent = {name = "Text",  
    bitrate = 128};
```

2.1.4 Constraints

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

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

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

The constraints in IVML will mostly rely on the relevant part of the syntax as well as on a large subset of the operations defined in OCL (c.f. Section 3 for a description of all operations). In IVML we use the constraint expression syntax of OCL, but omit the OCL contexts used to relate constraints to UML modelling elements. Similar to OCL, all elements defined in an IVML model will be accessible to constraints. Two examples for constraints are given below, one propositional and one first-order logic example using a quantifier:

- `(10 <= a and a <= 20) implies b == a;`
If `a` is in the range `(10; 20)` this implies that `b` must have the same value as `a`.
- `mySet->forAll(x | x > 100);`
All elements in `mySet` must be larger than 100

Constraints may be used in two distinct ways in IVML:

- **Standalone constraints:** Constraints are given as statements in a project or within a compound so that compound fields are directly accessible without qualification. As standalone constraints are used like statements, they end with a semicolon (as shown in the two examples above).
- **Embedded constraints:** One or more constraints are used as part of a statement, for example a `typedef`. Here the constraint is written in parenthesis and not ended by a semicolon.

Below we will discuss individual elements of constraints in IVML and, in particular, the difference (in particular regarding an adapted notation) to the related elements in OCL. Large parts of the remainder of this section are directly taken over from the OCL specification [4] and adapted to the IVML context.

Keywords

Keywords in IVML constraint expressions are reserved words. That means that the keywords cannot occur anywhere in an expression as the name of a decision variable or a compound. The list of keywords is shown below:

- **and**
- **def**
- **else**
- **endif**
- **if**
- **iff**
- **implies**
- **in**
- **let**
- **not**
- **or**
- **then**
- **xor**
- **null**

Prefix operators

IVML defines two prefix operators, the unary

- Boolean negation '**not**'.
- Numerical negation '**-**' which changes the sign of a Real or an Integer.

Infix operators

Similar to OCL, in IVML the use of infix operators is allowed. The operators '+', '-', '*', '/', '<', '>', '<>', '<=', '>=', '=', '==', '!=', and '<>' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

$a + b$

is conceptually equal to the expression:

$a . + (b)$

that is, invoking the "+" operation on *a* (the *operand*) with *b* as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators '<', '>', '<=', '>=', '<>', '**and**', '**or**', '**xor**', '**implies**', '**iff**' the return type is Boolean.

Please note that, while using infix operators, in IVML Integer is a subclass of Real. Thus, for each parameter of type Real, you can use Integer as the actual parameter. However, the return type will always be Real.

Equality and assignment operators (default logic)

In contrast to OCL, IVML provides two operators which are related to the equality of elements with different semantics, namely the default assignment '=' and the equality constraint operator '=='. We will explain the difference in this section.

Basically, a decision variable in IVML is considered as **undefined**, i.e., the variable does not have an effect on the instantiation. Constraints may explicitly refer to the undefined state via the operation "isDefined". Please note that for instantiation all (relevant) decision variables must be frozen (cf. Section 2.2.4.2) and that also undefined decision variables can be frozen.

A **default value** may be assigned to a variable. Default values can be used to define a basic configuration (a kind of basic profile) which applies to all products in the product line. A default value can be defined as part of the variable declaration² (using the '=', cf. Section 2.1.3) or in terms of an individual default assignment using the '=' operator. Default values may be changed by partial configuration (cf. Section 2.2.4.1), i.e. on the import path of a (hierarchical) variability model the default value of certain decision variables may be modified in order to adjust the basic profile, e.g., to a certain application setting or domain. However, a default value may only be modified (assigned or changed) once in a given model. This restriction is required due to the fact that IVML does not provide support to define the sequence of evaluations (except for imports and eval blocks, cf. Section 2.2.4.3).

As the '=' operator defines a default value which may be overridden, it is not possible to use that operator to express that a decision variable must have a certain value (under some conditions). This can be achieved using the equality operator '=='. Basically, the equality operator checks whether the left hand and the right hand operand have **equal values**. In two distinct cases, the equality operator **enforces the value** specified by the right hand operand. The cases are the

- Unconditional value constraint, e.g., `a == 5`.
- Conditional value constraint given as the right side of an implication, e.g., `c < 5 implies a == 5`.

In these two cases, the equality operator expresses that the left hand operand (an expression denoting a decision variable) must have the same value as the right hand operand. If the left hand operand contains a default value, then the default value will be overridden. However, if two expressions aim at enforcing different values for the same decision variable, the model becomes unsatisfiable.

Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot and arrow operations: '.' (for element and operation access) and '->' (to access collection operations such as **forAll** or **exists**).

² A decision variable declaration which defines a default value is semantically equivalent to a decision variable declaration without default value and a subsequent default assignment (somewhere) in the same model.

- unary **'not'** and unary minus **'-'**
- **'*'** and **'/'**
- **'+'** and binary **'-'**
- **'if-then-else-endif'**
- **'<'**, **'>'**, **'<='**, **'>='**
- **'=='** (equality), **'<>'**, **'!='** (alias for **'<>'**)
- **'and'**, **'or'** and **'xor'**
- Default assignment **'='**
- **'implies'**, **'iff'**

Parentheses **'('** and **')'** can be used to change the precedence of operators in expressions.

Type conformance

Type conformance in IVML constraints is inspired by OCL (cf. OCL section 7.4.5):

- AnyType is the common superclass of all types. All types comply with AnyType. However, AnyType is typically used for defining the built-in operations. The only value of AnyType is **null**, which explicitly makes a decision variable undefined.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies.
- The **refines** keyword induces a hierarchy of compounds where the subtypes are compliant to their parent types, i.e. the parent type may be replaced by each subtype.

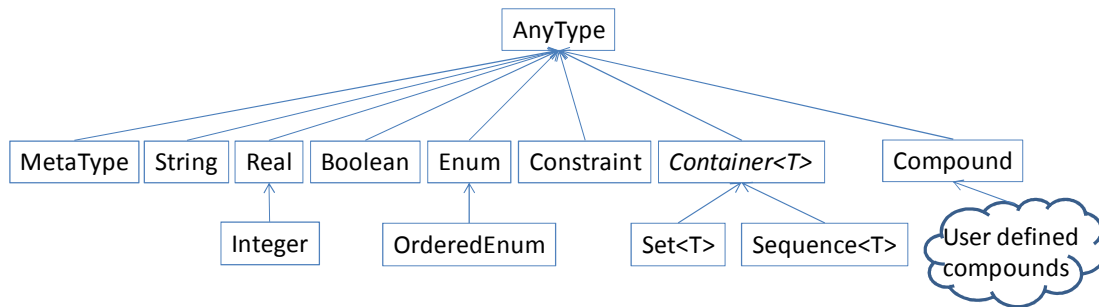


Figure 1: IVML type hierarchy

- Derived types are compliant to their base type as long as if no constraints were specified.
- MetaType is a specific type denoting types, e.g. to constrain types of elements within a collection.

Type operations

IVML provides the **isTypeOf()**, **isKindOf()** and **typeOf()** operations. The first two operations are similar to the related operations in OCL. The latter one returns the actual type (MetaType) of a decision variable, compound field or container element. MetaType allows equality and unequality comparisons. Currently, IVML neither supports re-typing or casting.

Enumeration Types

Enumerations literals are used just like qualified names, i.e. using a dot. For a certain enumeration type only the enumeration literals may be used with default assignment ('='), equality ('==') or unequality ('!=', '<>') operators. In case that ordinals are explicitly specified for enumeration literals, also relational operators ('<', '>', '<=', '>=') may be used.

Compound Types

Decision variable declarations defined within a compound can be accessed using the dot operator '.'.³

String Type

In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes. For details please refer to Section 3.

³ Please note that the current implementation of IVML accepts qualified and unqualified variable names within a compound while unqualified shall be default for denoting variables within the same compound. However, the current reasoning mechanism may not properly distinguish both cases so that a qualification with the compound name for variables denoting variables within a compound are required.

Configuration Type

A decision variable of type `Configuration` represents a variable constraint. Such a variable needs to be used somewhere in an IVML model in order to become active. Further statements or constraints may override the constraint in such a variable.

If-then-else-endif Expressions

The if-then-else-endif construct supports determining a value depending on a Boolean expression, similar to distinction of cases in mathematics. Exactly one expressions must be used within the `then` and `else` parts, both yielding the same type. The `else` part is not optional.

```
if contents[0].type == "video"

    then contents[0].bitrate

    else contents[0].highBitrate;
```

Let Expressions

Sometimes a sub-expression is used more than once in a constraint. The `let` expression allows one to define a variable that can be used in the constraint. We adjusted the notation to the IVML convention so that the type is stated before the name.

```
let Integer sumBitrate = bitrates->sum()

in sumBitrate <= 256;
```

A let expression may be included in any kind of OCL expression. It is only known within this specific expression.

User-defined operations

To enable the named reuse of (larger) constraint expressions, user-defined operations can be defined. The syntax of the operation definitions is similar to the Let expression, but each attribute and operation definition is prefixed with the keyword `def` as shown below. We adjusted the notation as IVML does not have OCL contexts (no colon after `def`) and that the type is stated before the name of the operation or parameter.

```
def Integer actualBitrate(Contents c) =

    if c.type == "video"

        then c.bitrate

        else c.highBitrate;
```

The name of an operation may not conflict with keywords, types, decision variables, etc. An user-defined operation may be used similar to build-in operations. Please note that prefix or infix use of user-defined operations is not supported.

```
actualBitrate(c) > 1024 implies highQuality == true;
```


Collection operations

IVML defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of constraining the contents of collections or projecting new collections from existing ones. However, we support only a relevant subset of the various notations in OCL. The different constructs are described in the following paragraphs. All collection operations (and only those) are accessed using the arrow-operator ‘->’.

In the first versions of OCL, all collection operations returned flattened collections, i.e. the entries of nested collections instead of the collections were taken over into the results. However, this was considered as an issue in OCL and does not fit to the explicit hierarchical nesting in IVML. Thus, collection operations in IVML do not apply flattening.

Sometimes an expression using operations results in a collection, while we are interested only in a special subset of the collection. The **select** operation specifies a subset of a collection:

```
collection->select(t|boolean-expression-with-t)

collection->select(ElementType t|
    boolean-expression-with-t)
```

Both expressions result in a collection that contains all the elements from `collection` for which the `boolean-expression-with-t` evaluates to true. Thereby, `t` is an iterator which will successively receive all values stored in `collection`. In the second form the type of the elements is explicitly specified. Note that the type of the iterator must comply with the element type of the collection. To find the result of this expression, for each element in `collection` the expression `boolean-expression-with-t` is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not.

Example:

```
/* Get all elements of the set "contents" with a
   "highBitrate" of less than 128 */

contents->select(t|t.highBitrate < 128);
```

The **reject** operation is identical to the **select** operation, but with **reject** we get the subset of all the elements of the collection for which the expression evaluates to False. The **reject** syntax is identical to the **select** syntax.

As shown in the previous section, the **select** and **reject** operations always result in a sub-collection of the original collection. When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a **collect** operation. The **collect** operation uses the same syntax as the **select** and **reject** and is written as one of:

```
collection->collect(t|boolean-expression-with-t)
```

```
collection->collect(ElementType t|
    boolean-expression-with-t)
```

Many times a constraint is needed on all elements of a collection. The **forAll** operation in IVML allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll(t|boolean-expression-with-t)

collection->forAll(ElementType t|
    boolean-expression-with-t)
```

Example:

```
/* None of the elements of the set "contents" must have a
"highBitrate" of greater than 512 */

contents->forAll(t|t.highBitrate <= 512);
```

The **forAll** operation has an extended variant in which more than one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a **forAll** on the Cartesian product of the collection with itself.

```
collection->forAll(t1, t2|
    boolean-expression-with-t1-and-t2)

collection->forAll(ElementType t1, t2|
    boolean-expression-with-t1-and-t2)
```

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The **exists** operation in IVML allows you to specify a Boolean expression that must hold for at least one object in a collection:

```
collection->exists(t|boolean-expression-with-t)

collection->exists(ElementType t|
    boolean-expression-with-t)
```

Depending on the type of the collection further related operation may be defined such as **isUnique**. Details will be given in Section 3 where we describe all operations in detail.

One special case of collection operation is to aggregate one value over all values in a collection by applying a certain expression or function. However, this comes close to the iterate operation in OCL. As we specifically target value aggregations define the **apply** operation while reusing the already known syntax:

```
collection->apply(t, ResultType r = initial|
    r = expression-with-t)
```

```
collection->apply(ElementType t, ResultType r = initial|  
    r = expression-with-t)
```

This operation initializes the result “iterator” `r` with the `initial` expression and applies the `expression-with-t` to each element in the collection. The result of `expression-with-t` is used to update successively the result “iterator”. Finally, the operation returns the value of `r` after processing the last element in `collection`. Please note that the result “iterator” is always defined using a specific type which, in turn, defines the result type of the `apply` operation.

Example:

```
/* Return the sum of all (default) bitrates of the  
elements of the set "contents" */  
  
contents->apply(t, Integer r| r == t.bitrate);
```

2.1.5 Configurations

The IVML does not differentiate between a configuration space and specific (product) configurations. Instead, a project can simultaneously describe or extend a configuration space and define a configuration. However, typically a project will provide a configuration space, while a different project may extend it, while providing configurations information for the initially specified configuration space. The set of decision variables and constraints of a project represent the set of all possible configurations. In addition, default values of decision variables as described in Section 2.1.3 define basic configurations and, thus, do not need to be further configured, but can be overwritten later as well. In addition, some values of decision variables can be derived using constraints. Any configuration, independent of where the values come from, must comply with the relevant constraints.

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

Example:

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

```
project contentSharing {  
  
    enum ContentType {text, video, audio, threeD, blob};  
  
    typedef Bitrate Integer with (Bitrate >= 128 and  
        Bitrate <= 256);  
  
    ContentType content;
```

```
    Bitrate contentBitrate = 128;  
  
    contentBitrate == 128 implies  
        content == ContentType.text;  
}
```

2.2 *Advanced Concepts of the INDENICA Variability Modelling Language*

This section describes advanced concepts of the IVML. We will describe how to assign additional attributes to modelling elements. This allows describing certain modelling elements in more detail, e.g. assigning meta-variability information as described in D2.1. We then augment the compound types introduced in Section 2.1.2.5 by extension and referencing concepts. Extension concepts will also be introduced for projects (cf. Section 2.1.1), which cover modularization aspects as well as facilitating project composition. We will describe advanced configuration concepts including partial configurations as well as “freezing” configurations. Finally, we will describe a lightweight concept for including DSLs as part of a variability model.

2.2.1 **Attributes**

In the IVML modelling elements can be attributed by further (orthogonal) configuration capabilities, e.g. to express meta-variability such as binding times. An attribute in IVML is basically a decision variable that is attached to another modelling element describing this element in more detail. Thus, an attribute may also have a default value and may be restricted by constraints (cf. Section 2.1.4). The impact of an attribute depends on the element it is attached to. In the IVML the following modelling elements can be attributed:

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

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

Syntax:

```
attribute Type name1 to name2;  
  
attribute Type name3 = value to name4;
```

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

- The **attribute** keyword indicates the definition of a new attribute.
- The expressions *Type name₁* and *Type name₃* correspond to the definition of a decision variable described in Section 2.1.3 while *name₁* and *name₃* are the identifiers of the new attributes⁴.
- The **to** keyword indicates the attachment of the new attribute on the left side to the element (*name₄*) denoted on the right side. Multiple names may be given separated by commas
- *name₄* may be one of the elements described above to which the attribute is attached.
- Optionally, a default value (*value*) can be assigned to the attribute by appending a value expression after *name₃*.

Example:

```
project contentSharing {  
    enum BindingTimes {configuration=0, compile=1,  
        runtime=2};  
  
    // Attaching an attribute to the entire project.  
  
    attribute BindingTimes binding = BindingTimes.compile  
        to contentSharing;  
}
```

Attributes can also be used in initializing expressions for containers and compounds. This is demonstrated in the fragment below:

```
compound Content {  
    String name;  
    Integer bitrate;  
}  
  
Content content;  
  
attribute BindingTimes binding = BindingTimes.compile  
    to content;
```

⁴ Due to technical reasons, currently attributes must not start with 'v' or 'e'.

```
content = {name="Video", bitrate=128,  
          name.binding=BindingTimes.compile,  
          bitrate.binding=BindingTimes.runtime};
```

However, assigning the same value for a certain attribute for a given set of decision variables may increase the perceived complexity of the model as similar assignments are repeated.

Example:

```
project contentSharing {  
  
    enum BindingTimes {configuration=0, compile=1,  
                      runtime=2};  
  
    // Attaching an attribute to the entire project.  
  
    attribute BindingTimes binding = BindingTimes.compile  
    to contentSharing;  
  
    enum Colors {black, white};  
  
    Bitrate contentBitrate = 128;  
  
    contentBitrate.binding = BindingTimes.configuration;  
  
    Colors backgroundColor = Colors.black;  
  
    backgroundColor.binding = BindingTimes.configuration;  
  
    // go on with several variables and different binding  
  
    // times  
  
}
```

IVML provides the assign construct as syntactic sugar to simplify the mass-assignment of values to attributes and to visually group the model elements with same (initial) attribute assignment. However, the variables “declared” in the assign block actually are part of the containing element, in the example below the project `contentSharing`. An assign block can also be used within compounds, it may even be nested in other assign blocks if needed or multiple attributes may be given in comma-separated fashion in the parenthesis of an assign block. As an assign block is technically translated into individual assignment constraints (`=`) as stated as a generic constraint in the parenthesis of an assign block.

Example:

```
project contentSharing {
```

```

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

// Attaching an attribute to the entire project.

attribute BindingTimes binding = BindingTimes.compile
    to contentSharing;

enum Colors {black, white};

assign (binding = BindingTimes.configuration) to {

    Bitrate contentBitrate = 128;

    Colors backgroundColor = Colors.black;

    // go on with the variables of the same binding time
}

}

```

2.2.2 Advanced Compound Modelling

In Section 2.1.2.5 we introduced the compound types to group multiple types into a single named unit. In this section, we will extend the modelling of compound types by refinement and referencing concepts. Refinement allows extending existing compound types by additional elements, yielding a new (extended) compound type. Referencing enables the definition of references to other elements like other compounds.

2.2.2.1 Extending Compounds

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

Syntax:

```

compound Name1 refines Name2 {

    // Define additional elements.

}

```

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

- The **compound** keyword indicates the definition of a new compound type.
- The identifier *Name₁* defines the name of the new compound type.
- The **refines** keyword indicates that the new compound type (*Name₁*) is an extension of a previously defined compound type (*Name₂*).

- The set of elements surrounded by curly brackets defines the additional elements that make up the extensions to the inherited elements of compound *Name₂*.

Example:

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

compound Content {

    String name;

    Integer bitrate;

}

/* A new compound type that refines the previous compound
type. "ExternalContent" will subsume all elements of
"Content" and all additional elements defined below. */

compound ExternalContent refines Content {

    String contentPath;

    String accessPassword;

}
```

2.2.2.2 Referencing Elements

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

Syntax:

```
project name1 {

    compound Name2 {

        Type name3;

        ...

    }

    // Declaration of a new reference.

    refTo(Name2) Name4;
```



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

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

- The **refTo** keyword indicates the definition of a new reference.
- *Name₂* defines the referenced element (type).
- *Name₄* is an identifier and defines the name of the new reference. In the IVML a reference is type, thus, the identifier for a new reference starts with a capital letter.
- The **refBy** keyword indicates the configuration of a reference (the configuration of the referenced element respectively).
- *Name₄* is an identifier that defines the reference to be configured.
- The syntax for configuring a reference depends on the type of the referenced element (see Section 2.1.3 for the syntax for assigning values to variables of a specific type). In the case above, we use “.”-notation to configure a single element of a referenced compound type.

Example:

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

```
compound Container {  
  
    String name;  
  
    ...  
  
}
```

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

```
compound Content {  
  
    String name;  
  
    Integer bitrate;
```

```
// Declaration of a reference to the Container compound.

refTo(Container) myContainer;

// Configuration of the above reference.

refBy(myContainer).name = "ContentContainer";

}
```

2.2.3 Advanced Project Modelling

In Section 2.1.1, we introduced the concept of projects (**project**) as the top-level element in each IVML-model. In this section, we extend the modelling capabilities of the IVML regarding projects in three ways: first, we describe versioning of projects that enables the definition of the current state of evolution of a project. This concept correlates with the second concept: project composition. This introduces the capability of deriving new projects based on definitions in other projects and explicitly excluding certain projects from the composition. As part of this version information can be used. The third concept is project interface. The concepts of project composition and project interfaces support effective modularization and reuse of projects and, thus, configuration spaces.

2.2.3.1 Project Versioning

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

Syntax:

```
project name {

    // Definition of a version for this project

    version vNumber.Number;

    ...

}
```

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

- The **version** keyword indicates the definition of a new version for the project *name*.

- *vNumber.Number* defines the actual version of the project (here only two parts prefixed by a “v”). At least one number must be given and no restriction holds on the amount of sub-version numbers.

Example:

```
project contentSharing {  
    version v1.0;  
    ...  
}
```

2.2.3.2 Project Composition

The IVML supports the composition of different projects. This is closely related to multi software product lines [8] and product populations [9]. Project composition allows to effectively reusing existing projects by using these projects within other projects. This also supports the decomposition of large variability models as semantically related parts can be defined in individual projects. The complete project then uses these (sub-) projects to define the combined project. In the IVML the following keywords are introduced for project composition:

- **import**: this keyword indicates the use of a project. An imported project is evaluated before import, thus an import acts as an implicit eval. This keyword allows using certain elements of a project by reference. If a project contains explicit interfaces (see below), the specific interface, which is used, must be given.

However, multiple projects with identical names and versions may exist in a file system⁵, in particular in hierarchical product lines. Thus, project imports are determined according to the following **hierarchical import convention**, i.e. starting at the (file) location of the importing project (giving precedence to imports in the same file) the following locations are considered in the given sequence: The same directory, then contained directories (closest directories are preferred) and finally containing directories (also here closest directories are preferred). Similar to Java class paths, additional model paths⁶ may be considered in addition to the immediate file hierarchy.

- **conflicts**: this keyword indicates incompatibility among projects. All projects (names) followed by this keyword cannot be used in combination with the project that defines this conflict expression. This is also checked for indirectly used projects. Also project names in conflicts are resolved according to the hierarchical import convention defined above.

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

⁵ The implementation of the tool support decides whether the entire file system or a subtree is considered. In EasY-Producer, currently the entire active workspace is considered.

⁶ The actual implementation is already prepared for model paths. Depending on the actual use we will include model paths into the user-level of the tool support.

project introduced in Section 2.2.3.1. Note, that versions restrictions are no fully-fledged constraints and only the relational operators ' $<$ ', ' $>$ ', ' $<=$ ', ' $>=$ ' as well as the equality operators ' $=$ ', ' $==$ ', ' $<>$ ', ' $!=$ ' may be used here. Please note that version numbers start with "v" (cf. Section 2.2.3.1).

Syntax:

```

project name1 {

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

    import name2;

    // Accessing elements of a project.

    name2::element;

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

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

}

```

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

- The keyword **import** indicates that the entities, which are made available by the project or interface *name*₂ will be available within the current project.
- For disambiguation the elements of *name*₂ can be accessed using the "::*element*"-notation to express qualified names. If there is no ambiguity, they can be used directly.
- The keyword **conflicts** indicates incompatibility of project *name*₁ with project *name*₃.
- Optionally, version-expressions can be combined with the keywords **import** and **conflicts** using the **with** keyword. This defines specific versions of other projects to be imported into the current project or conflicting with the current project.
- A version expression includes the version-information of a project (cf. Section 2.2.3.1), a relation operator and a version number or a version-information of another project. In addition, logical operators can be used to concatenate simple version-expressions to define ranges of versions.

Example:

```
project application{

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

    String name;

}


project targetPlatform{

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

    version v1.5;

    String name;

}


project contentSharing{

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

    import application;

    import targetPlatform
        with (targetPlatform.version >= v1.3);

    // Accessing the elements of the sub-projects.

    application::name = "myApp";

    targetPlatform::name = "myPlatform";

}
```

2.2.3.3 Project Interfaces

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

Interfaces in a project define all elements of a project, not part of the interface, as private and, thus, make them invisible to the outside. This is indicated by the **interface** keyword within a project. In order to access any elements they need to be declared as parameters of the interface. This can be done by exporting existing variables (using the **export** keyword) or by declaring new parameter variables. As a special characteristic of the IVML, it is also possible to define multiple interfaces for the same project. This is different from other variability modelling languages like the CVL [6].

Importing a project (cf. Section 2.2.3.2) that includes interfaces allows the importing project to access only the parameters defined in the interface. All other elements of the project are not visible to the importing project.

Syntax:

```
project name1 {  
  
    // Definition of a new interface.  
  
    interface Name2 {  
  
        /* Denotes the export of an existing decision variable  
        of the project name1. */  
  
        export name3;  
  
        ...  
    }  
  
    /* Declaration of a (private) decision variable. This  
    variable is exported by the interface Name2. */  
  
    Type name3;  
}
```

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

- The keyword **interface** indicates the definition of a new interface of the project *name*₁. Interfaces must occur at the beginning of a project before decision variable or type definitions.
- The keyword **export** indicates the export of the following decision variable *name*₃.

Example:

```
project application {  
  
    // This will define an interface for this project.
```

```
interface MyInterface {  
    export name, appType;  
}  
  
// Declaration of (private) decision variables.  
  
String name;  
  
String appType;  
  
Integer bitrate;  
  
// Definition of a constraint.  
  
appType == "Video" implies bitrate == 256;  
}  
  
project contentSharing{  
    /* This will import the interface "MyInterface" of  
    project "application". */  
  
    import application::MyInterface;  
  
    /* Only the parameters of the interfaces are accessible.  
    "application::bitrate" yields an error. As long as the  
    variable names are unambiguous, the fully qualified must  
    not be used. */  
  
    name = "myApp";  
  
    appType = "Video";  
}
```

2.2.4 Advanced Configuration

In Section 2.1.5, we introduced the configuration concept of the IVML. In this section, we will extend this concept to partial configuration. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring only parts of the project. The set of all configuration steps typically yield a full configuration of the entire project. We will further introduce the concept of persistent (parts of) configurations. We call this “freezing”. Freezing (parts of) configurations defines these parts to be persistent. Persistent parts cannot be

changed anymore in further configuration steps. Finally, we will describe how (parts of) configurations can be evaluated independently from other parts of the configuration. This allows deriving additional configuration values based on existing configurations using the constraints and value propagation.

2.2.4.1 Partial Configurations

The IVML supports partial configurations. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring only parts of the project. The set of all configuration steps typically yields a full configuration of the entire project. The configuration of a part of a project may also be reconfigured by the next configuration step (cf. the concept of default values, which we introduced in Section 2.1.3). For example, a service provider may define a (pre-) configuration of the provided service, while a service consumer may reconfigure his service to satisfy his specific needs.

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

Example:

```
project application{

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

    String name = "Application";

}

project targetPlatform{

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

    String name;

}

project contentSharing{

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

    import application;

    import targetPlatform;
```



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

application::name = "myApp";

targetPlatform::name = "myPlatform";

}
```

2.2.4.2 Freezing Configurations

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

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

Freezing an undefined variable v leaves v undefined so that v does not have an effect. In particular, v may be changed afterwards and v may be part of a configuration implicitly disabling some instantiation.

Syntax:

```
project  $name_1$  {

    // Definition of new compound type

    compound  $Name_2$  {

        Type  $name_3$ ;

        Type  $name_4$ ;

    }

    /* Declaration of a new decision variable of the above
type */

     $Name_2$   $name_6$ ;
```

```
/* Freezing the configuration of the decision variable
except element name4. */

name6.name3 = value1;

freeze {

    name6;

} but (name6.name4)

}
```

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

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

Example:

```
project application {

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

    compound ContentType {

        String contentName;

        Integer bitrate;

    }

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

    ContentType appContent;

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

    appContent.contentName = "Text";

    freeze {
```

```
        appContent;  
    } but (appContent.bitrate)  
}
```

2.2.4.3 Partial Evaluation

The IVML provides a concept for the evaluation of configurations. This is indicated by the keyword **eval**. The explicit declaration of *nested eval* structures can be used to structure the definition of the variables and thus reduces the search-space during constraint-evaluation. By default, the top-level **eval** structure is the containing project, i.e., at the end of a project definition an implicit **eval** occurs as the project is the topmost eval. **eval** structures on the same nesting level do not imply a sequence of evaluation as this is true for the constraints in a project.

Currently, an eval statement may only contain constraints, i.e., variables are project global and no variables can be defined in an eval (this may change in future, then variables would be propagated from inside the eval the outside eval or project).

Syntax:

```
/* Evaluate a constraint that defines the relation between  
two variables of the same type. This leads to the  
assignment of the variable values to the unassigned  
variable upon exit of the scope of the eval-statement.  
Note that this eval is evaluated before any other  
constraint in the project is evaluated.*/
```

```
eval {  
  
    name1 = name2;  
  
}
```

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

Example:

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

```
    }

    // Declaration of a decision variable of the above type.
    ContentType appContent;

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

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

project targetPlatform{

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

    String name;

    Integer bitrate;

}

project contentSharing{

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

    import application;

    import targetPlatform;

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

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

```
}
```

2.2.5 Including DSLs

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

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

Syntax⁷:

```
DSL(stopString, prefix, dslInterpreter) %

// here goes the DSL

DSL%;
```

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

- The *stopString* identifier⁸ is a string used for uniquely identifying the end of the DSL in combination with the **DSL** keyword. The part between the opening **DSL** keyword (excluding its parameters in parentheses) and the closing **DSL** keyword (marked by the *stopString*) is not analyzed by the IVML tools but passed to an external DSL interpreter for evaluation.
- The *prefix* identifier is a string identifying a DSL-specific prefix for IVML identifiers denoting decision variables. When passing the DSL code to the DSL specific tools, all occurrences of decision variables marked by the *prefix* are replaced by actual values for the individual decisions.
- The *dslInterpreter* identifier is a string containing, for example, a file name or an URI specifying the concrete DSL tool which is responsible for evaluating the instantiated DSL code, i.e. after substituting occurrences of decision variables.

Example:

```
project application {
```

⁷ Technically, a DSL fragment is implemented as an expression of AnyType. AnyType is the common supertype in the type system of OCL and IVML. However, in IVML AnyType cannot be used in subexpressions, i.e. a DSL fragment is written as a standalone expression statement while the use within an expression is syntactically but not semantically correct.

⁸ Due to technical reasons implementing this concept in xText, the *stopString* is now fixed to **DSL%**. The parameter *stopString* is kept as parameter for legacy reasons but may be subject to removal in future versions.

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

Integer bitrate = 128;

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

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

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

    ...

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

    ... $bitrate ...

DSL%;
}
```

3 Constraints in IVML

In this section we will describe syntax and semantics of the IVML constraint sublanguage. In Section 3.1 we will describe the constraint language and in Section 3.2 the built-in operation which can be used within constraint expressions.

3.1 *IVML constraint language*

In this section we will define the syntax and the semantics of the IVML constraint language. As constraints in IVML heavily rely on OCL, most of the content in this section is taken from OCL [4] and adjusted to the notational conventions and the semantics of IVML.

3.1.1 Keywords

Keywords in IVML constraint expressions are reserved words. That means that the keywords cannot occur anywhere in an expression as the name of a decision variable or a compound. The list of keywords is shown below:

- **and**
- **def**
- **else**
- **endif**
- **if**
- **iff**
- **implies**
- **in**
- **let**
- **not**
- **or**
- **then**
- **xor**

3.1.2 Prefix operators

IVML defines two prefix operators, the unary

- Boolean negation '**not**'.
- Numerical negation '**-**' which changes the sign of a Real or an Integer.

3.1.3 Infix operators

Similar to OCL, in IVML the use of infix operators is allowed. The operators '+', '-', '*', '.', '/', '<', '>', '<=>', '<=' '>=' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

a + b

is conceptually equal to the expression:

a . + (b)

that is, invoking the “+” operation on a (the *operand*) with b as the parameter to the operation. The infix operators defined for a type must have exactly one parameter. For the infix operators ‘<’, ‘>’, ‘<=’, ‘>=’, ‘<>’, ‘and’, ‘or’, ‘xor’, ‘implies’, ‘iff’ the return type must be Boolean.

Please note that, while using infix operators, in IVML integer is a subclass of real. Thus, for each parameter of type real, you can use integer as the actual parameter. However, the return type will always be real. We will detail the operations on basic types in Section 3.4.

3.1.4 Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot and arrow operations: ‘.’ (for element and operation access) and ‘->’ (to access collection operations such as **forAll** or **exists**).
- unary ‘not’ and unary minus ‘-’
- ‘*’ and ‘/’
- ‘+’ and binary ‘-’
- ‘if-then-else-endif’
- ‘<’, ‘>’, ‘<=’, ‘>=’
- ‘==’ (equality), ‘<>’, ‘!=’ (alias for ‘<>’)
- ‘and’, ‘or’ and ‘xor’
- Default assignment ‘=’
- ‘implies’, ‘iff’

Parentheses ‘(’ and ‘)’ can be used to change precedence.

3.1.5 Datatypes

All datatypes defined in IVML including the user-defined ones such as compounds, restricted types or attributes are available to the constraint language and may be used in constraint expressions. Below, we give some specific notes on the use of datatypes, in particular in relation to OCL.

- In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes. For details please refer to Section 3.
- Enumerations literals are used just like qualified names, i.e. using a dot. For a certain enumeration type only the enumeration literals may be used with assignment (‘=’), equality (‘==’) or inequality (‘!=’, ‘<>’) operators. In case that ordinals are explicitly specified for enumeration literals, also relational operators (‘<’, ‘>’, ‘<=’, ‘>=’) may be used.
- Decision variable declarations defined within a compound can be accessed using the dot operator ‘.’.

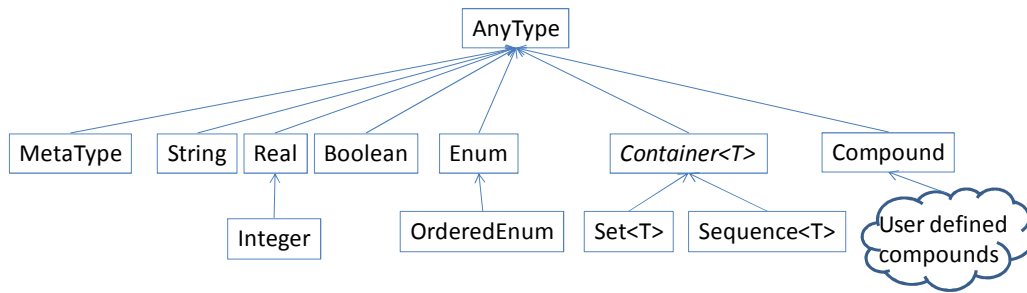


Figure 2: IVML type hierarchy

3.1.6 Type conformance

Type conformance in IVML constraints is inspired by OCL (cf. OCL section 7.4.5):

- AnyType is the common superclass of all types. All types comply with AnyType. However, AnyType is typically used for defining the built-in operations.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies.
- The **refines** keyword induces a hierarchy of compounds where the subtypes are compliant to their parent types, i.e. the parent type may be replaced by each subtype.
- Derived types are compliant to their base type as long as if no constraints were specified.
- MetaType is a specific type denoting types, e.g. to constrain types of elements within a collection.

3.1.7 Type operations

IVML provides the following type-specific operations: **isTypeOf()**, **isKindOf()** and **typeOf()**. The first two operations are similar to the related operations in OCL. The latter one returns the actual type (MetaType) of a decision variable, compound field or container element. MetaType allows equality and unequality comparisons. In addition, the collections provide the operations **typeSelect** and **typeReject** which select elements from a collection according to their actual type based on the **isTypeOf** operation. Currently, IVML neither supports re-typing or casting.

3.1.8 Side effects

IVML is designed as a modelling and configuration language for Software Product Lines. As a configuration language, an assignment of values to decision variables is mandatory. Thus, in contrast to OCL, some constraint expressions in IVML may lead to side effects in terms of value assignments ('='). Please note that all operations except for assignments are free of side effects (similar to OCL).

3.1.9 Undefined values

Basically, variables are undefined in order to enable partial configuration. Unless a default value ('=') or a value (via the constraint operator '==') is assigned. Due to undefined variables, some expressions will, when evaluated, have an undefined value. During evaluation, undefined (sub-) expressions are ignored.

3.1.10 If-then-else-endif Expressions

The if-then-else-endif construct supports determining a value depending on a Boolean expression, similar to distinction of cases in mathematics. Exactly one expressions must be used within the `then` and `else` parts, both yielding the same type. The `else` part is not optional.

```
if contents[0].type == "video"

    then contents[0].bitrate

    else contents[0].highBitrate;
```

3.1.11 Let Expressions

Sometimes a sub-expression is used more than once in a constraint. The `let` expression allows one to define a variable that can be used in the constraint. We adjusted the notation to the IVML convention so that the type is stated before the name.

```
let Integer sumBitrate = bitrates->sum()

in sumBitrate <= 256;
```

A `let` expression may be included in any kind of OCL expression. It is only known within this specific expression.

3.1.12 User-defined operations

To enable the named reuse of (larger) constraint expressions, user-defined operations can be defined. The syntax of the operation definitions is similar to the `Let` expression, but each attribute and operation definition is prefixed with the keyword `def` as shown below. We adjusted the notation as IVML does not have OCL contexts (no colon after `def`) and that the type is stated before the name of the operation or parameter.

```
def Integer actualBitrate(Contents c) =

    if c.type == "video"
```

```
    then c.bitrate  
    else c.highBitrate;
```

The name of an operation may not conflict with keywords, types, decision variables, etc. An user-defined operation may be used similar to build-in operations. Please note that prefix or infix use of user-defined operations is not supported.

```
actualBitrate(c) > 1024 implies highQuality == true;
```

3.1.13 Collection operations

IVML defines many operations on the collection types. These operations are specifically meant to enable a flexible and powerful way of constraining the contents of collections or projecting new collections from existing ones. However, we support only a relevant subset of the various notations in OCL. The different constructs are described in the following paragraphs. All collection operations (and only those) are accessed using the arrow-operator ‘->’.

In the first versions of OCL, all collection operations returned flattened collections, i.e. the entries of nested collections instead of the collections were taken over into the results. However, this was considered as an issue in OCL and does not fit to the explicit hierarchical nesting in IVML. Thus, collection operations in IVML do not apply flattening.

Sometimes an expression using operations results in a collection, while we are interested only in a special subset of the collection. The **select** operation specifies a subset of a collection:

```
collection->select(t|boolean-expression-with-t)  
  
collection->select(ElementType t|  
    boolean-expression-with-t)
```

Both expressions result in a collection that contains all the elements from `collection` for which the `boolean-expression-with-t` evaluates to true. Thereby, `t` is an iterator which will successively receive all values stored in `collection`. In the second form the type of the elements is explicitly specified. Note that the type of the iterator must comply with the element type of the collection. To find the result of this expression, for each element in `collection` the expression `boolean-expression-with-t` is evaluated. If this evaluates to true, the element is included in the result collection, otherwise not.

Example:

```
/* Get all elements of the set "contents" with a  
"highBitrate" of less than 128 */  
  
contents->select(t|t.highBitrate < 128);
```

The **reject** operation is identical to the select operation, but with reject we get the subset of all the elements of the collection for which the expression evaluates to False. The reject syntax is identical to the select syntax.

As shown in the previous section, the select and reject operations always result in a sub-collection of the original collection. When we want to specify a collection which is derived from some other collection, but which contains different objects from the original collection (i.e., it is not a sub-collection), we can use a **collect** operation. The **collect** operation uses the same syntax as the select and reject and is written as one of:

```
collection->collect(t|boolean-expression-with-t)

collection->collect(ElementType t|
    boolean-expression-with-t)
```

Many times a constraint is needed on all elements of a collection. The **forAll** operation in IVML allows specifying a Boolean expression, which must hold for all objects in a collection:

```
collection->forAll(t|boolean-expression-with-t)

collection->forAll(ElementType t|
    boolean-expression-with-t)
```

Example:

```
/* None of the elements of the set "contents" must have a
"highBitrate" of greater than 512 */

contents->forAll(t|t.highBitrate <= 512);
```

The **forAll** operation has an extended variant in which more than one iterator is used. Both iterators will iterate over the complete collection. Effectively this is a **forAll** on the Cartesian product of the collection with itself.

```
collection->forAll(t1, t2|
    boolean-expression-with-t1-and-t2)

collection->forAll(ElementType t1, t2|
    boolean-expression-with-t1-and-t2)
```

Many times one needs to know whether there is at least one element in a collection for which a constraint holds. The **exists** operation in IVML allows you to specify a Boolean expression that must hold for at least one object in a collection:

```
collection->exists(t|boolean-expression-with-t)

collection->exists(ElementType t|
    boolean-expression-with-t)
```

Depending on the type of the collection further related operation may be defined such as `isUnique`. Details will be given in Section 3 where we describe all operations in detail.

One special case of collection operation is to aggregate one value over all values in a collection by applying a certain expression or function. However, this comes close to the iterate operation in OCL. As we specifically target value aggregations define the **apply** operation while reusing the already known syntax:

```
collection->apply(t, ResultType r = initial|
    r = expression-with-t)

collection->apply(ElementType t, ResultType r = initial|
    r = expression-with-t)
```

This operation initializes the result “iterator” `r` with the `initial` expression and applies the `expression-with-t` to each element in the collection. The result of `expression-with-t` is used to update successively the result “iterator”. Finally, the operation returns the value of `r` after processing the last element in `collection`. Please note that the result “iterator” is always defined using a specific type which, in turn, defines the result type of the `apply` operation.

Example:

```
/* Return the sum of all (default) bitrates of the
elements of the set "contents" */

contents->apply(t, Integer r | r == t.bitrate);
```

3.2 Built-in operations

Similar to OCL, in the IVML constraint language all operations are defined on individual IVML types and can be accessed using the “.” operator, such as `set.size()`. However, this is also true for the equality, relational and mathematical operators but they are typically given in alternative infix notation, i.e. `1 + 1` instead of `1.+(1)`. Further, the unary negation is typically stated as prefix operator. Iterative collection operations such as `forAll` are the only⁹ operations in IVML which are accessed by “->”. However, IVML also defines some specific operations which are also listed with their defining type below.

In this section, we denote the actual type on which an individual operation is defined as the *operand* of the operation (called *self* in OCL). The parameters of an operation are given in parenthesis. Further, similar to the declaration of decision variables in IVML, we use in this section the Type-first notation to describe the signatures of the operation.

⁹ This is due to technical restrictions realizing IVML with Xtext.

3.3 *Internal Types*

3.3.1 **AnyType**

AnyType is the most common type in the IVML type system. All types in IVML are subclasses of AnyType, i.e. they are type compliant and inherit the operations listed below.

- **Boolean == (AnyType a)**
True if the *operand* is the same as *a*. This operation is interpreted as a value assertion if it is used standalone (empty implication) or on the right side of an implication. It is interpreted as an equality test if used on the left side of an implication.
- **Boolean <> (AnyType a)**
True if the *operand* is different from *a*.
- **Boolean != (AnyType a)**
True if the *operand* is a different object from *a*. Alias for !=.
- **MetaType typeOf ()**
The type information of the actual type.
- **Boolean isTypeOf (MetaType type)**
True if the *type* and the actual type of *operand* are the same. This operation can be seen as an alias for typeOf() == *type*.
- **Boolean isKindOf (MetaType type)**
True if *type* is either the direct type or one of the supertypes of the actual type of the *operand*.

3.3.2 **MetaType**

MetaType represents the actual type of an object such as a specific user-defined container. Currently, MetaType inherits all operations from AnyType except for the typeOf, isTypeOf and isKindOf operations.

3.4 *Basic Types*

3.4.1 **Real**

The basic type Real represents the mathematical concept of real following the Java range restrictions for double values. Note that Integer is a subclass of Real, so for each parameter of type Real, you can use an integer as the actual parameter.

- **Real + (Real r)**
The value of the addition of *self* and the *operand*.
- **Real - (Real r)**
The value of the subtraction of *r* from the *operand*.
- **Real * (Real r)**
The value of the multiplication of the *operand* and *r*.
- **Real - ()**
The negative value of the *operand*.
- **Real / (Real r)**

The value of the *operand* divided by *r*. Leads to an evaluation error if *r* is equal to zero.

- **Real abs()**
The absolute value of the *operand*.
- **Integer floor ()**
The largest integer that is less than or equal to the *operand*.
- **Integer round()**
The integer that is closest to *the operand*. When there are two such integers, the largest one.
- **Real max (Real r)**
The maximum of the *operand* and *r*.
- **Real min (Real r)**
The minimum of the *operand* and *r*.
- **Boolean < (Real r)**
True if the *operand* is less than *r*.
- **Boolean > (Real r)**
True if the *operand* is greater than *r*.
- **Boolean <= (Real r)**
True if the *operand* is less than or equal to *r*.
- **Boolean >= (Real r)**
True if the *operand* is greater than or equal to *r*.
- **Boolean = (Real r)**
Assigns the value *r* to the variable *operand* and returns *true*¹⁰.

3.4.2 Integer

The standard type Integer represents the mathematical concept of integer following the Java range restrictions for integer values. Note that Integer is a subclass of Real.

- **Integer - ()**
The negative value of the *operand*.
- **Integer + (Integer i)**
The value of the addition of the *operand* and *i*.
- **Integer - (Integer i)**
The value of the subtraction of *i* from the *operand*.
- **Integer * (Integer i)**
The value of the multiplication of the *operand* and *i*.
- **Real / (Integer i)**
The value of the *operand* divided by *i*. Leads to an evaluation error if *i* is equal to zero.
- **Integer abs()**
The absolute value of the *operand*.
- **Integer div (Integer i)**
The number of times that *i* fits completely within the *operand*.

¹⁰ The Boolean return type is required as stand-alone constraints must be of Boolean type. The result of an assignment operation is always *true*.

- **Integer mod (Integer i)**
The result is the *operand* modulo *i*.
- **Integer max (Integer i)**
The maximum of the *operand* and *i*.
- **Integer min (Integer i)**
The minimum of the *operand* and *i*.
- **Boolean < (Integer i)**
True if the *operand* is less than *i*.
- **Boolean > (Integer i)**
True if the *operand* is greater than *i*.
- **Boolean <= (Integer i)**
True if the *operand* is less than or equal to *i*.
- **Boolean >= (Integer i)**
True if the *operand* is greater than or equal to *i*.
- **Boolean = (Integer i)**
Assigns the value *i* to the operand and returns *true*¹⁰.

3.4.3 Boolean

The basic type Boolean represents the common true/false values.

- **Boolean or (Boolean b)**
True if either *self* or *b* is true.
- **Boolean xor (Boolean b)**
True if either *self* or *b* is true, but not both.
- **Boolean and (Boolean b)**
True if both *b1* and *b* are true.
- **Boolean not ()**
True if *self* is false and vice versa.
- **Boolean implies (Boolean b)**
True if *self* is false, or if *self* is true and *b* is true. The rightmost implication is interpreted as an assertion of the right side of the expression. Further implications on the left side of an implication as well as implication in a Boolean expression are just evaluated to a Boolean value.
- **Boolean iff (Boolean b)**
Shortcut for (a.implies(b) and b.implies(a)).
- **Boolean = (Boolean b)**
Assigns the value *b* to the operand and returns *true*¹⁰.

3.4.4 String

The standard type String represents strings, which can be ASCII.

- **Integer size ()**
The number of characters in the *operand*.
- **String concat (String s)**
The concatenation of the *operand* and *s*.
- **String substring (Integer lower, Integer upper)**

The sub-string of the *operand* starting at character number *lower*, up to and including character number *upper*. Character numbers run from 0 to *size()*.

- **Boolean matches (String r)**
Returns whether the *operand* matches the regular expression *r*. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether `mail` is a valid e-mail-address:
`mail.matches([\\w]*@[\\w]*.[\\w]*);`
- **Boolean substitutes (String r, String s)**
Replaces all occurrences of the regular expression *r* in the *operand* by *s*. Regular expressions are given in the Java regular expression notation. For example, the following operation will substitute the occurrence of "@" with "{at}" in an e-mail-address:
`mail.substitutes("@", "{at}");`
- **Integer toInteger ()**
Converts the *operand* to an Integer value.
- **Real toReal ()**
Converts the *operand* to a Real value.
- **Boolean = (String s)**
Assigns the value *s* to the operand and returns *true*¹⁰.

3.5 Enumeration Types

Enumerations allow the definition of sets of named values.

3.5.1 Enum

Enums inherit all operations from AnyType and adds the following operation:

- **Boolean = (Enum e)**
Assigns the value *e* to the operand and returns *true*¹⁰.

3.5.2 OrderedEnum

In contrast to Enums, individual ordinal values for the literals in an OrderedEnum are specified. Thus, an OrderedEnum defines a (total) ordering on its literals so that further operations in addition to those defined for Enum are available.

- **Boolean < (OrderedEnum l)**
True if the *operand* is less than the ordinal value of the literal *l*.
- **Boolean > (OrderedEnum l)**
True if the *operand* is greater than the ordinal value of the literal *l*.
- **Boolean <= (OrderedEnum l)**
True if the *operand* is less than or equal to the ordinal value of the literal *l*.
- **Boolean >= (OrderedEnum l)**
True if the *operand* is greater than or equal to the ordinal value of the literal *l*.

3.5.3 Constraint

The basic type `Constraint` represents variable constraints. In addition to the operations provided by `AnyType`, the `Constraint` type provides the following operations:

- **Boolean = (Constraint c)**
Assigns the constraint *c* to the operand and returns *true*¹⁰.

3.6 Collection Types

This section defines the operation of the collection types. The two IVML collections `Set` and `Sequence` are both subtypes of the abstract collection type `Collection`. Each collection type is actually a template type with one parameter. 'T' denotes the parameter. A concrete collection type is created by substituting a type for the T. So a collection of integers is referred in IVML by `setOf(Integer)`.

3.6.1 Collection

`Collection` is the abstract superclass of all collections in IVML.

- **Integer size ()**
The number of elements in the collection *operand*.
- **Boolean includes (T object)**
True if *object* is an element of *operand*, false otherwise.
- **Boolean excludes (T object)**
True if *object* is not an element of *operand*, false otherwise.
- **Integer count (T object)**
The number of times that *object* occurs in the collection *operand*.
- **Boolean isEmpty ()**
Is the *operand* the empty collection?
- **Boolean notEmpty ()**
Is the *operand* not the empty collection?
- **Boolean isDefined()**
Returns whether (a variable of) the *operand* is defined, i.e. that an instance was already assigned.
- **T sum()**
The addition of all elements in the *operand*. Elements must be of a type supporting the + operation (Integer or Real).
- **T product()**
The multiplication of all elements in the *operand*. Elements must be of a type supporting the * operation (Integer or Real).
- **T min()**
The minimum of all elements in the *operand*. Elements must be of a type supporting the < operation (Integer or Real).
- **T max()**
The maximum of all elements in the *operand*. Elements must be of a type supporting the > operation (Integer or Real).
- **T avg()**

The average of all elements in the *operand*. Elements must be of a type supporting the / operation (Integer or Real).

- **Boolean forAll (Iterators | expression)**
Results in true if *expression* evaluates to true for each element in the *operand* collection.
- **Boolean exists (Iterators | expression)**
Results in true if *expression* evaluates to true for at least one element in the *operand* collection.
- **Boolean isUnique (Iterator | expression)**
Results in true if *expression* evaluates to a different value for each element in the *operand* collection; otherwise, result is false. *isUnique* may have at most one iterator variable.
- **T any (Iterator | expression)**
Returns any element in the *source* collection for which *expression* evaluates to true. If there is more than one element for which *expression* is true, one of them is returned. *any* may have at most one iterator variable.
- **Boolean one (Iterator | expression)**
Results in true if there is exactly one element in the *operand* collection for which *expression* is true. *one* may have at most one iterator variable.
- **Collection<T> collect (Iterator | expression)**
The Collection of elements that results from applying *expression* to every member of the *source* set. *collect* may have at most one iterator variable.
- **Collection<T> select (Iterator | expression)**
The sub-collection for which expression is true. *select* may have at most one iterator variable.
- **Boolean reject (Iterator | expression)**
The sub-collection for which expression is false. *reject* may have at most one iterator variable.
- **<R> apply (Iterator, R result | result = expression)**
Applies the given expression to the operand collection using the specified iterator and stores the result in the last iterator (used here as a local variable declaration) which is returned as the result of this operation. Expression shall use the result “iterator” for aggregating values. Apply may have at most one iterator variable and needs to specify the result “iterator”.

3.6.2 Set

The Set is the mathematical set. It contains elements without duplicates. Set inherits the operations from Collection.

- **Boolean == (Set<T> s)**
Evaluates to true if *operand* and *s* contain the same elements.
- **Set<T> union (Set<T> s)**
The union of *operand* and *s*.
- **Set<T> intersection (Set<T> s)**
The intersection of *operand* and *s* (i.e., the set of all elements that are in both *operand* and *s*).
- **Set<T> excluding (T object)**

The set containing all elements of *operand* without *object*.

- **Set<T> including (T object)**
The set containing all elements of *operand* plus *object*.
- **Set<T> asSet ()**
A Set identical to *operand*. This operation exists for convenience reasons.
- **Sequence<T> asSequence ()**
A Sequence that contains all the elements from *operand*, in undefined order.
- **Set<T> typeSelect (MetaType T)**
Results the subset of elements from *operand* which are of type T.
- **Set<T> typeReject (MetaType T)**
Results the subset of elements from *operand* which are not of type *T*.
- **Boolean = (Set<T> s)**
Assigns the value *s* to the operand and returns *true*¹⁰.

3.6.3 Sequence

A sequence is a collection where the elements are ordered. An element may be part of a sequence more than once. Sequence inherits the operations from Collection.

- **Boolean == (Sequence<T> s)**
Evaluates to true if *operand* and *s* contain the same elements.
- **Sequence<T> union (Sequence<T> s)**
The union of *operand* and *s*.
- **Set<T> asSet ()**
The Set containing all the elements from *operand*, with duplicates removed.
- **Sequence<T> asSequence ()**
The Sequence identical to the *operand* itself. This operation exists for convenience reasons.
- **T at (Integer i)**
The *i*-th element of the sequence *operand*. Valid indices run from 0 to *size()*-1.
- **T [] (Integer i)**
The *i*-th element of the sequence *operand*. This operation is an alias for *at*. Valid indices run from 0 to *size()*-1.
- **T first ()**
The first element in *operand*.
- **T last()**
The last element in *operand*.
- **Sequence<T> append (T object)**
The sequence of elements, consisting of all elements of *operand*, followed by *object*.
- **Sequence<T> prepend(T object)**
The sequence consisting of *object*, followed by all elements in *operand*.
- **Sequence<T> insertAt(Integer index, T object)**
The sequence consisting of *operand* with *object* inserted at position *index*. Valid indices run from 0 to *size()*-1.
- **Integer indexOf(T object)**
The index of object *object* in the sequence *operand*.

- **Sequence<T> typeSelect (MetaType T)**
Results the subset of elements from *operand* which are of type *T*.
- **Sequence<T> typeReject (MetaType T)**
Results the subset of elements from *operand* which are not of type *T*.
- **Boolean = (Sequence<T> s)**
Assigns the value *s* to the operand and returns *true*¹⁰.

3.7 Compound Types

A compound type groups multiple types into a single named unit. A compound inherits all its operations from *AnyType*. Access to variable declarations within a compound are specified using “.”. Using the type name of the compound on the left side of a “.” is a shortcut for an all-quantification on all instances of that compound. In addition, it defines the following operation:

- **Boolean isDefined()**
Returns whether (a variable of) the *operand* is defined, i.e. that an instance was already assigned.
- **Boolean = (Compound c)**
Assigns the value *c* to the operand and returns *true*¹⁰.

4 IVML Grammar

In this section we depict the actual grammar for IVML. The grammar is given in six sections (basic modeling concepts, basic types and values, advanced modeling concepts, basic constraints, advanced constraints and terminals) in terms of a simplified xText¹¹ grammar (close to ANTLR¹² or EBNF). Simplified means, that we omitted technical details in xText used to properly generate the underlying EMF model as well as trailing “;” (replaced by empty lines in order to support readability). Please note that some statement-terminating semicolons are optional in order to support various user groups each having individual background in programming languages.

4.1 *Basic modeling concepts*

VariabilityUnit:

Project*

Project:

```
'project' Identifier '{'
    VersionStmt?
    ImportStmt*
    ConflictStmt*
    InterfaceDeclaration*
    ProjectContents
'}' ';'?
```

ProjectContents:

```
(Typedef
    | VariableDeclaration
    | Freeze
    | Eval
    | ExpressionStatement
    | AttributeTo
    | OpDefStatement
    | AttrAssignment
)*
```

¹¹ <http://www.eclipse.org/Xtext/>

¹² <http://www.antlr.org>

ExpressionBlock:

```
'{'
    ExpressionStatement+
'}' ';'?
```

Typedef:

```
TypedefEnum
| TypedefCompound
| TypedefMapping
```

TypedefEnum:

```
'enum' Identifier
'{'
    TypedefEnumLiteral (',' TypedefEnumLiteral)*
'}'
TypedefConstraint?
```

TypedefEnumLiteral:

```
Identifier ('=' NumValue)?
```

TypedefCompound:

```
'compound' Identifier ('refines' Identifier)?
'{'
    (VariableDeclaration
    | ExpressionStatement
    | AttrAssignment)*
'}' ';'?
```

TypedefMapping:

```
'typedef' Identifier Type TypedefConstraint? ';'?
```

TypedefConstraint:

```
'with' '(' Expression (',' Expression)* ')'
```

VariableDeclaration:

```
Type VariableDeclarationPart (',' VariableDeclarationPart)* ';'?
```

VariableDeclarationPart:

```
Identifier ('=' Expression)?
```

```
DerivedType:
    (
        'setOf'
        | 'sequenceOf'
        | 'refTo'
    )
    '(' Type ')'
```

4.2 *Basic types and values*

```
BasicType:
    'Integer'
    | 'Real'
    | 'Boolean'
    | 'String'
    | 'Constraint'
```

```
Type:
    BasicType
    | QualifiedName
    | DerivedType
```

```
NumValue:
    NUMBER
```

```
QualifiedName:
    (Identifier '::' (Identifier '::')*)? Identifier
```

```
AccessName:
    ('.' Identifier)+
```

```
Value:
    NumValue
    | STRING
    | QualifiedName
    | ('true' | 'false')
    | ('refby' '(' Identifier ')')
```


4.3 *Advanced modeling concepts*

AttributeTo :

```
'attribute' Type VariableDeclarationPart 'to' Identifier
(',' Identifier)*';'
```

AttrAssignment:

```
'assign'
 '(' AttrAssignmentPart (',' AttrAssignmentPart)* ')' 'to'
 '{'
   (VariableDeclaration | ExpressionStatement | AttrAssignment)+
 '}' ';'?
```

AttrAssignmentPart:

```
Identifier '=' LogicalExpression
```

Freeze:

```
'freeze' '{'
   FreezeStatement+
 '}' ('but' FreezeButList)? ';'?
```

FreezeStatement:

```
QualifiedName AccessName? ';' 
```

FreezeButList:

```
'(' FreezeButExpression (',' FreezeButExpression)* ')'
```

FreezeButExpression:

```
QualifiedName AccessName? '*'?
```

Eval:

```
'eval' ExpressionBlock
```

InterfaceDeclaration:

```
'interface' Identifier '{'
   Export*
 '}' ';'?
```

Export:

```
'export' Identifier (',' Identifier)* ';' 
```

ImportStmt:

```
'import' Identifier (':' Identifier)?  
(  
    'with' '(' VersionedId (',' VersionedId)* ')' )?  
)? ';' 
```

ConflictStmt:

```
'conflicts' Identifier  
(  
    'with' '(' VersionedId (',' VersionedId)* ')' )?  
)? ';' 
```

VersionedId:

```
Identifier '.version' VersionOperator VERSION
```

VersionOperator:

```
'==' | '>' | '<' | '>=' | '<=' | '<>' | '!='
```

VersionStmt:

```
'version' VERSION ';' 
```

DslContext:

```
'DSL' '(' STRING ',' STRING ',' STRING ')'  
DSL_CONTENT
```

4.4 *Basic constraints*

ExpressionStatement:

```
Expression ';' 
```

Expression:

```
LetExpression  
| ImplicationExpression  
| CollectionInitializer  
| DslContext
```

ImplicationExpression:

AssignmentExpression ImplicationExpressionPart*

ImplicationExpressionPart:

ImplicationOperator AssignmentExpression

ImplicationOperator:

'implies' | 'iff'

AssignmentExpression:

LogicalExpression AssignmentExpressionPart?

AssignmentExpressionPart:

'=' (LogicalExpression | CollectionInitializer)

LogicalExpression:

EqualityExpression LogicalExpressionPart*

LogicalExpressionPart:

LogicalOperator EqualityExpression

LogicalOperator:

'and' | 'or' | 'xor'

EqualityExpression:

RelationalExpression EqualityExpressionPart?

EqualityExpressionPart:

EqualityOperator (RelationalExpression | CollectionInitializer)

EqualityOperator:

'==' | '<>' | '!='

RelationalExpression:

AdditiveExpression RelationalExpressionPart?

RelationalExpressionPart:

RelationalOperator AdditiveExpression

RelationalOperator:

'>' | '<' | '>=' | '<=' | '<>' | '!='

AdditiveExpression:

MultiplicativeExpression AdditiveExpressionPart*

AdditiveExpressionPart:

AdditiveOperator MultiplicativeExpression

AdditiveOperator:

'+' | '-'

MultiplicativeExpression:

UnaryExpression MultiplicativeExpressionPart?

MultiplicativeExpressionPart:

MultiplicativeOperator UnaryExpression

MultiplicativeOperator:

'*' | '/'

UnaryExpression:

UnaryOperator? PostfixExpression

UnaryOperator:

'not' | '-'

PostfixExpression:

(FeatureCall Call* ExpressionAccess?)
| PrimaryExpression

Call:

'.' FeatureCall
| '->' SetOp
| '[' Expression '']'

FeatureCall:

Identifier '(' ActualParameterList? ')'

SetOp:

```
Identifier
'(' Declarator Expression? ')'
```

Declarator:

```
Declaration (';' Declaration)* '|'
```

Declaration:

```
Identifier (',' Identifier)* (':' Type)? ('=' Expression)?
```

ActualParameterList:

```
Expression (',' Expression)*
```

ExpressionAccess:

```
'.' Identifier Call* ExpressionAccess?
```

PrimaryExpression:

```
(
    Literal
    | '(' Expression ')'
    | IfExpression
    | 'refBy' '(' Identifier ')'
)
Call*
ExpressionAccess?
```

CollectionInitializer:

```
QualifiedName?
'{'
    ExpressionList?
'{'
```

ExpressionList:

```
ExpressionListEntry (',' ExpressionListEntry)*
```

ExpressionListEntry:

```
(Identifier ('.' Identifier)? '=')?
(LogicalExpression | LiteralCollection)
```

Literal:
Value

4.5 *Advanced constraints*

LetExpression:
'let' Type Identifier '=' Expression 'in' Expression

IfExpression:
'if' Expression 'then' Expression 'else' Expression 'endif'

OpDefStatement:
'def' Type Identifier '(' OpDefParameterList ')' '
'=' Expression ';'

OpDefParameterList:
(OpDefParameter (',' OpDefParameter)*)?

OpDefParameter:
Type Identifier ('=' Expression)?

4.6 *Terminals*

Identifier:
ID | VERSION | EXPONENT

terminal VERSION:
'v' ('0'..'9')+ ('.' ('0'..'9')+)*

terminal ID:
('a'..'z' | 'A'..'Z' | '_') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*

terminal NUMBER:
'-'?
(('0'..'9')+ ('.' ('0'..'9')* EXPONENT?)?
| '.' ('0'..'9')+ EXPONENT?
| ('0'..'9')+ EXPONENT)

terminal EXPONENT:
('e' | 'E') ('+' | '-')? ('0'..'9')+

terminal STRING :

```
'"' (
    '\\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'"'|'\\\' ) | !('\\\'|'"')
)* '"'
|
'"' (
    '\\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'"'|'\\\' ) | !('\\\'|'"')
)* '"'
```

terminal DSL_CONTENT:

```
'%' -> 'DSL%'
```

terminal ML_COMMENT:

```
'/*' -> '*/'
```

terminal SL_COMMENT:

```
'//\' !('\'n'|\'r')* (\'r'? \'n')?
```

terminal WS:

```
(' '|\'t'|\'r'|\'n')+
```

terminal ANY_OTHER:

```
.
```

References

- [1] K. Bak, K. Czarnecki, and A. Wasowski. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In B. Malloy, S. Staab, and M. van den Brand, editors, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE '10)*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2010.
- [2] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-Based Feature Modelling Language. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '10)*, pages 159–162, 2010.
- [3] E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, editors. *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*. IEEE, 2011.
- [4] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: <http://www.omg.org/docs/formal/06-05-01.pdf>.
- [5] Object Management Group, Inc. (OMG). Unified Modeling Language: Superstructure version 2.1.2. Specification v2.11 2007-11-02, Object Management Group, November 2007. Available online at: <http://www.omg.org/docs/formal/2007-11-02.pdf>.
- [6] Object Management Group, Inc. (OMG). Common Variability Language (CVL), 2010. OMG initial submission. Available on request.
- [7] Mark-Oliver Reiser. Core Concepts of the Compositional Variability Management Framework (CVM). Technical Report 2009/16, Technische Universität Berlin, 2009. Available online at <http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/>.
- [8] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '10)*, pages 123–130, 2010.
- [9] Rob van Ommering. *Building Product Populations with Software Components*. PhD thesis, University of Groningen, 2004.

INDENICA Variability Implementation Language: Language Specification

Version 0.7

(corresponds to VIL bundle versions 0.0.7)

Software Systems Engineering (SSE)

University of Hildesheim

31141 Hildesheim

Germany

Abstract

Creating domain-specific service platforms requires the capability of customizing and configuring service platforms according to the specific needs of a domain. In this document we provide a novel approach for variability implementation. We focus on how to implement selected customization and configuration options in service (platform) ecosystems in a generic way focusing on the specification of the instantiation process. This enables domain engineers to define their specific instantiation process in a declarative way without the need for implementation of specific tool components such as instantiators.

In this document we specify the concepts of the INDENICA variability implementation language (VIL) for specifying how customization and configuration options in service (platform) ecosystems can be turned into (instantiated) artefacts.

Version

0.5	15. June 2013	first version derived from D2.2.2
0.6	8. August 2013	revised concepts
0.7	26. September 2013	revisions based on actual implementation

Document Properties

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

Table of Contents

1	Introduction	7
2	The INDENICA Variability Implementation Approach	8
3	The VIL Languages.....	10
3.1	INDENICA Variability Build Language.....	11
3.1.1	Reserved Keywords.....	11
3.1.2	Scripts.....	12
3.1.3	Version	13
3.1.4	Imports.....	14
3.1.5	Types	15
3.1.5.1	Basic Types	15
3.1.5.2	Configuration Types	15
3.1.5.3	Artefact Types	16
3.1.5.4	Container Types	17
3.1.6	Variables.....	17
3.1.7	Externally Defined Values of Global Variables.....	18
3.1.8	Rules.....	19
3.1.8.1	Variable Declarations.....	22
3.1.8.2	Expressions.....	22
3.1.8.3	Calls	22
3.1.8.4	Operating System Commands	25
3.1.8.5	Iterated Execution.....	25
3.1.8.6	Join Expression.....	26
3.2	VIL Template Language	27
3.2.1	Reserved Keywords.....	27
3.2.2	Template	27
3.2.3	Version	29
3.2.4	Imports.....	29
3.2.5	Functional Extension.....	30
3.2.6	Types	30
3.2.7	Variables.....	30
3.2.8	Sub-Templates (defs)	31
3.2.8.1	Variable Declaration	32

3.2.8.2	Expression Statement	32
3.2.8.3	Alternative.....	33
3.2.8.4	Switch.....	33
3.2.8.5	Loop.....	35
3.2.8.6	Content	35
3.3	VIL Expression Language	37
3.3.1	Reserved Keywords.....	37
3.3.2	Prefix operators	37
3.3.3	Infix operators.....	37
3.3.4	Precedence rules.....	38
3.3.5	Datatypes	38
3.3.6	Type conformance	39
3.3.7	Side effects.....	39
3.3.8	Undefined values	39
3.3.9	Collection operations.....	39
3.4	Built-in operations	40
3.4.1	Internal Types	40
3.4.1.1	AnyType	40
3.4.1.2	Type.....	40
3.4.2	Basic Types	40
3.4.2.1	Real.....	41
3.4.2.2	Integer.....	41
3.4.2.3	Boolean	42
3.4.2.4	String	42
3.4.3	Container Types	43
3.4.3.1	Collection	43
3.4.3.2	Set	44
3.4.3.3	Sequence.....	44
3.4.3.4	Map	44
3.4.4	Configuration Types	45
3.4.4.1	IvmlElement	45
3.4.4.2	EnumValue	46
3.4.4.3	DecisionVariable	46

3.4.4.4	Attribute	46
3.4.4.5	IvmlDeclaration	46
3.4.4.6	Configuration	46
3.4.5	Built-in Artefact Types and Artefact-related Types	47
3.4.5.1	Path	47
3.4.5.2	JavaPath	48
3.4.5.3	Project	48
3.4.5.4	Text.....	49
3.4.5.5	Binary	49
3.4.5.6	Artifact	49
3.4.5.7	FileSystemArtifact	50
3.4.5.8	FolderArtifact	50
3.4.5.9	FileArtifact.....	50
3.4.5.10	VtlFileArtifact	51
3.4.5.11	XmlFileArtifact.....	51
3.4.6	Built-in Instantiators	52
3.4.6.1	VIL Template Processor.....	52
3.4.6.2	Blackbox Instantiators.....	53
4	VIL Grammars.....	54
4.1	VIL Build Language Grammar.....	54
4.2	VIL Template Language Grammar	56
4.3	Common Expression Language Grammar.....	57
	References	63

Table of Figures

Figure 1: Overview of the VIL type system	38
---	----

1 Introduction

This document specifies the INDENICA variability implementation language (VIL) in terms of a living document, which describes the most current version of the language based on discussions with the partners and experiences made during the project.

VIL consists two languages: a build process description language and a template language. The focus of the VIL build language is on instantiating a whole service platform in terms of a software product line, while the template language focuses on the instantiation and creation of individual artefacts. Both VIL languages are based on an explicit and extensible artefact model as well as a tight integration with the INDENICA variability modelling language IVML [3].

The remainder of this language specification is structured as follows: in Section 2, we will briefly introduce the VIL approach. In Section 3, we will define the syntax and semantics of the aforementioned VIL languages, their common expression language as well as the underlying type system (including the artefact model and the IVML integration). Finally, in Section 4, we will provide the grammars of the VIL languages as a reference.

2 The INDENICA Variability Implementation Approach

In this section, we describe the concepts of the INDENICA Variability Implementation Language (VIL). VIL is designed to realize the instantiation of artefacts in a generic way, i.e., using a specification-based approach instead of relying on domain- or product-line-specific implemented instantiation mechanisms. A more detailed discussion of the approach idea and its benefits for product line engineering can be found in D2.2.2 [5].

The VIL is more than a single language. It consists of two languages and requires the understanding of additional core concepts:

- **Artefact meta-model:** Everything that can be instantiated (transformed or generated) is regarded as an artefact. The VIL approach relies on an artefact meta-model as its foundation. The artefact meta-model (or often artefact model for short) describes what operations can be performed on certain types of artefacts, 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 Deliverable D2.2.2 [5] as it mainly modifies Java bytecode and, thus, it is easier to realize (at least some part of it) in an usual programming language such as Java, i.e., from the point of view of VIL as a black box. Another example is a programming language compiler or a linker, which should not be re-developed using VIL but simply reused. In case of legacy product lines, an existing instantiator may be called or wrapped into a VIL extension.
- **VIL build language:** This is the main part of the VIL language as it binds all other pieces together. This is used to define individual production strategies, i.e., to relate artefacts and instantiation mechanisms, to combine production strategies in terms of rules and to specify the execution of the rules. Basically, it is a rule-based programming language as a foundation for describing product line instantiation processes.

VIL and its sublanguages are tightly integrated with IVML, i.e., IVML identifiers and configuration values can be directly used in VIL. From a more general point of view, VIL and its sublanguages rely on existing, practically proven concepts such as build rules 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 as we experienced in our analysis of related technologies.

Thus, we reuse and extend existing concepts to apply it to variability realization and created the VIL as a completely new language along with a novel implementation.

3 The VIL Languages

In this section, we will describe the two (sub) languages of VIL, i.e., the VIL build language and the VIL template language, as well as their main concepts. Due to the nature of both languages as variability implementation languages, they share a common type system as well as a common expression language.

The **extensible VIL type system** is the foundation for both VIL sub languages. The type system consists of basic types such as Integer or Boolean, configuration-related types realizing the integration with an IVML [3] variability model, artefact-related types implementing the artefact (meta) model, implicit types representing instantiators and derived types such as containers. In particular, the type system is extensible, i.e., additional or refining artefact types or instantiators can easily be added (in terms of Java classes). If compared with an object-oriented language, the artefact types can be considered as classes, the operations as methods, individual artefacts as instances and the execution of artefact operations as method calls. However, the instantiators can be more aptly compared to transformation rules as they are first of all rule-based and second operate on the artefact model, but are themselves not part of it. The **common VIL expression language** represents a wide range of expressions from simple calculations over artefact operation and instantiator calls up to rather complex composite expressions. The expression language relies on the operations and operators provided by the VIL type system.

The two VIL languages are realized on top of the common expression language. Both languages follow a textual approach to the specification of artefact and product instantiation and support batch processing. Our definition of the syntax of the VIL languages draws upon typical concepts used in programming languages, in particular Java, build languages such as *make*, template languages such as *xtend* as well as expressions inspired by IVML and the Object Constraint Language (OCL) [6]. We adapt these concepts as needed to provide additional operations required in variability implementation, such as the integration with a variability model or an explicit artefact model.

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

- The syntax as well as the examples will be illustrated in `Courier New`.
- **Keywords** will be highlighted using bold font.
- *Elements and expressions* that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number.

- Statements will be separated using semicolon “;” (most other language concepts may optionally be ended by a semicolon).
- Different types of brackets will be used to indicate lists “()”, sets “{ }”, etc. This is closely related to the Java programming language.
- We will indicate comments using “//” and “/* . . . */” (cf. Java).

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

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

In Section 3.1, we will describe the specific concepts of the VIL build language which is responsible for specifying the overall variability instantiation process of an entire (hierarchical) product line. In Section 3.2, we will describe the concepts of the VIL template language, which provides the means to describe the instantiation of a single (textual) artefact. Basic concepts of the VIL build and the VIL template language are rather similar (also to IVML) in order to simplify learning and application of these languages. In Section 3.3, we will detail the common expression language which is part of both, the VIL build and the VIL template language. In particular, we will detail the type system, i.e., the built-in types, their individual operations and the default instantiators that are part of the VIL implementation.

3.1 *INDENICA Variability Build Language*

In this section, we describe the concepts and language elements of the VIL build language in detail. This language aims at specifying the variability instantiation process of a whole (hierarchical) product line (as opposed to the instantiation of a specific artefact type covered by the VIL template language).

However, the VIL build language focuses on the implementation and instantiation of variabilities rather than on the entire build process of a whole system. Thus, the VIL build language is intended as an extension to existing build languages, i.e., it shall be integrated with those languages rather than replacing them.

3.1.1 Reserved Keywords

In the VIL build language, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by the keywords of the common VIL expression language given in Section 3.3.1.

- **@advice**
- **const**
- **exclude**
- **extends**
- **execute**

- `import`
- `join`
- `load`
- `properties`
- `protected`
- `version`
- `vilScript`
- `with`

3.1.2 Scripts

In the VIL build language a script (`vilScript`) is the top-level element. This element is mandatory as it identifies the production strategies to be applied to derive an instantiated product. The definition of a script requires a name, as a basis for referring among VIL build scripts and a parameter list specifying the expected information from the execution environment such as the actual configuration or the projects to work on. In order to realize the necessary capabilities required for implementing the hierarchical product line capabilities of the EASy producer tool, at least the source project to instantiate from, the target project to be instantiated and the actual variability configuration must be passed to a VIL build script.

Basically, VIL may refer to all visible configuration settings in a variability configuration, more precisely to the actual values of frozen decision variables (and their underlying structure). In order to make this integration explicit, these decision variables may be directly referred in VIL by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. This may complicate the development of VIL scripts as actually unknown identifiers will at least lead to a warning. To support the domain engineer in specifying valid build scripts, VIL provides the `advice` annotation specifying the name for the IVML models used in the VIL build scripts. Qualified names resolvable via the `advice` annotation do not lead to warnings in the VIL editor. As an explicit version number may be stated for VIL scripts (akin to IMVL models), also advices and model imports may be version-constrained.

Optionally, a VIL script may extend another VIL script, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

Syntax:

```
//imports
@advice(ivmlName)
vilScript name (parameterList) extends name1 {
    //optional version specification
    //loading of variable values from an external source
    //variable definitions
    //rule declarations
```

}

Description of syntax: the definition of a build script consists of the following elements:

- First, all referenced scripts must be imported. We will detail the import syntax in Section 3.1.4.
- Optional advices declaring the underlying variability models.
- The keyword **vilscript** defines that the identifier *name* is defined as a new build script with contained production strategies.
- The parameter list denotes the arguments to be passed to a VIL script for execution. When executed by EASy, at least the source project(s), the target project, and the variability configuration must be passed in. Source and target project may be identical in case of (traditional) in-place instantiation. However, further parameters may be given upon an explicit invocation from an external call, e.g., an integration with a build language such as ANT or Maven.
- A VIL build script may optionally extend an existing (imported) VIL script. This is expressed by **extends** *name₁*, whereby *name₁* denotes the name of the extending script.
- Production strategies are described within the curly brackets.

Example:

```
@Advice(YMS)
vilscript YMSBuild(Project source, Configuration config,
                  Project target){
    /* Go on with the production strategies for YMS here */
}
```

Please note that the types shown above such as `Project` or `Configuration` will be explained in detail in the next section. Further, a build script for multi-product lines may require a container of projects (see Section 3.1.5.4), while single project parameter is sufficient for a traditional product line build.

3.1.3 Version

Akin to IMVL, VIL build specifications may optionally be tagged with an explicit version number in order to support product line evolution. Evolution of software may yield updates to projects, IVML models and build scripts so that scripts of different versions may exist and need to be clearly distinguished.

Syntax:

```
// Declaration of the version of a VIL build script.
version vNumber.Number;
```

Description of Syntax: A version statement consists of the following elements:

- The **version** keyword indicates a version declaration. At maximum one version declaration may be given in a VIL build file at the very first position within a VIL build script.
- *vNumber.Number* defines the actual version of the project (here only two parts prefixed by a “v”). At least one number must be given and no restriction holds on the amount of sub-version numbers.
- A version statement ends with a semicolon.

Example:

```
vilscript YMSBuild(Project source, Configuration config,  
                  Project target){  
    version v0.1.4;  
    ...  
}
```

3.1.4 Imports

The production strategies for a variability instantiation build process may be defined in a single VIL build script or may be reused from other (existing) build scripts. Therefore, VIL build scripts may be imported. In order to support also the evolution of product line build specifications, VIL allows the specification of version-restricted imports. Imports make the production strategies defined in the specified build file accessible to the importing build script.

Syntax:

```
// Unconstraint and constraint imports.  
import name;  
import name with (version op vNumber.Number);
```

Description of Syntax: An import¹ of a build scripts consists of the following elements:

- Importing a build script starts with the keyword **import**. Multiple imports may be given in a VIL build file directly at the beginning of the script file.
- *name* (given in terms of a VIL identifier) refers to the name of the build script to be imported. However, multiple scripts with identical names and versions may exist in a file system, in particular in hierarchical product lines. Thus, imports are determined according to the following **hierarchical import convention**, i.e., starting at the (file) location of the importing script (giving precedence to imports in the same file) the following locations are considered in the given sequence: The same directory, then

¹ Actually, this syntax differs from IVML due to technical reasons in xText.

contained directories (closest directories are preferred) and finally containing directories (also here closest directories are preferred). Similar to Java class paths, additional script paths may be considered in addition to the immediate file hierarchy.

- An optional restriction of the import in terms of versions. This is indicated by the keyword **with** followed by a parenthesis containing the restrictions. A restriction is stated by the keyword **version**, a comparison operator (**=**, **>**, **<**, **>=**, **<=**) and a version number. An and-clause of multiple restrictions may be given in the parenthesis separated by commas.
- An import statement ends with a semicolon.

Example:

```
vilscript YMSBuild(Project source, Configuration config,  
                  Project target){  
    version v0.1.4;  
    import generics with (version >= v1.12);  
}
```

3.1.5 Types

Basically, the VIL build language is a statically typed language with partially postponed type checking at runtime as we will detail below. Thus, the VIL build language provides a set of formal types to be used in variable declarations or parameter lists. We distinguish between basic types, configuration types, artefact types, and container types.

3.1.5.1 Basic Types

The basic types in the VIL build language correspond to the basic types of IVML, i.e., Boolean (**Boolean**), integer (**Integer**), real (**Real**) and string (**String**) with their usual meaning.

Boolean constants are given in terms of the keywords **true** and **false**. Integer constants are stated as usual numbers not containing a **“.**” or an exponential notation. Real constants must contain the floating-point separator **“.**” or may be given in exponential notation. Strings are either given in quotes or in apostrophs and may contain the usual escape sequences including those for line ends, quotes and apostrophs.²

3.1.5.2 Configuration Types

A configuration type denotes the representation of IVML configuration elements in VIL. However, due to the nature of VIL, we need only access to the configuration and the structure of an IVML model rather than to all modelling capabilities. Thus, VIL provides a specific set of built-in configuration types. The actual instance of a

² Strings delimited by quotes may contain apostrophs, strings delimited by apostrophs may contain quotes.

configuration is passed into a VIL build script in terms of a script parameter. Configuration types cannot be directly created in a VIL script and must not modify the underlying IVML model.

The entry point to a configuration in terms of an IVML model is the type `Configuration`. It provides access to all frozen decision variables and attributes. In particular, `Configuration` allows creating projections of a given configuration in order to simplify further processing. Further, it provides access to IVML type declarations such as compounds or enumerations and their value. This is represented by the `IvmlElement` and its subtypes. An `IvmlElement` represents IVML concepts in a generic way and provides access to its (qualified) name, its (qualified) type name and the configured value. Specific subtypes of `IvmlElement` are `DecisionVariable`, `Attribute` and `IvmlDeclaration`, each providing with more specific operations as we will discuss in detail in Section 3.4.4.

3.1.5.3 Artefact Types

Artefact types represent the different categories of artefacts used in the artefact model. Some artefact types are built-in and part of the VIL implementation, while further types can be defined in terms of an extension of the artefact model. In this section, we will discuss only the predefined types. Please refer to the EASY developers guide on how to define more specific artefact types (as well as how to integrated instantiators implemented in a programming language).

The type `Project` is a mapping of a physical project (Eclipse) into VIL and provides related operations such as mapping paths between the source and the target project for instantiation.

A `Path` is a predefined type of the VIL artefact model although it is not an artefact by itself. A `Path` represents a relative file system path and may possibly contain wildcards. A path is specified in terms of a `String` in VIL and is automatically converted into a `Path` or an artefact instance depending on the actual use. In more detail, paths are specified according to the ANT [9] conventions, i.e., using the slash as path separator and wildcards for patterns. The following wildcards are supported: `?` for a single character (excluding the path separator), `*` for multiple characters (excluding the path separator) and `**` for (sub) path matches.

`Artifact`³ is the most common artefact type and root of the VIL artefact hierarchy. The predefined `Artifacts` have also predefined methods. For example, they allow to delete the artefact (if possible at all), or to obtain access to its plain textual or binary representation. VIL provides a set of built-in artefact types such as `FileArtifact` and `FolderArtifact` which are both `FileSystemArtifacts`. Further, VIL provides more specific artefact types such as the `VtlFileArtifact` representing VIL template files (see Section 3.4.5) or the `XmlFileArtifact` representing parsed XML files with a substructure of specialized fragment artefacts such as `XmlElement` or `XmlAttribute`. Please note that artefact instances are assigned in a polymorphic way, i.e., while a `FileArtifact` may be specified as type in a VIL script, it may actually contain a more specific type.

³ We adopted US English in the implementation of VIL.

3.1.5.4 Container Types

VIL provides three container types, sequences (keyword `sequenceOf`), sets (keyword `setOf`) and associative containers (keyword `mapOf`). Container types are generic with respect to their content type(s) and, similarly to IVML, the content type must be stated explicitly, such as `sequenceOf(Integer)` or `setOf(DecisionVariable, FileArtifact)`.

Sequences may contain an arbitrary number of elements of a given element type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. In sequences, elements can be accessed by their position in the container using an index (`[index]`). In VIL, indexes start at zero and run until the number of elements in the container minus one (as in Java and many other languages). Collections typically occur as results of operations, rule, or instantiator executions. In addition, they can be explicitly initialized using type-compatible expressions of the appropriate dimension as shown follows

```
sequenceOf(Integer) someNumbers = {1, 2, 3, 4, 5};
setOf(Integer, Integer) somePairs = {{1, 2}, {3, 4}};
```

A `Map` represents an associative container in VIL, i.e., a container which relates a keys to associated values. In particular, it allows retrieving the value assigned to a key via the `get` operation and the `[]`-Operator (`[key]`). Basically, associative containers are intended to simplify the translation of IVML-identifiers to implementation-specific identifiers in individual artefacts. Therefore, VIL associative containers can be explicitly initialized in terms of key-value-pairs using type-compatible expressions

```
setOf(String, String) idTranslation
= {{ "nrOfProcessors", "procCnt"}, {"nrOfNodes", "nodeCnt"} };
```

VIL supports a set of operations specific for container types, e.g., excluding, projecting, or collecting elements in a container, etc. We will introduce the full set of operations in Section 3.4.3.

3.1.6 Variables

A variable provides name-based access to a value of a certain type (see Section 3.1.5), similar to variables in programming languages.

In VIL, the value of a variable can be modified at any time (in contrast to build languages such as ANT [9] where a value of a property can be set only once). In addition, a variable may be declared to be constant so that a value can be set only once and not be modified afterwards. Variables may be of global scope, i.e., directly defined within a VIL script or they may be local (within rules, see Section 3.1.6).

Syntax:

```
// Declaration of a variable.
Type variableName1;
Type variableName2 = value;
const Type constantName = value;
```

Description of Syntax: The declaration of variables consists of the following elements:

- The **Type** defines the type of the variable being declared.
- The identifiers *variableName₁*, *variableName₂* and *constantName* are the names of the declared variable or constant, respectively.
- The optional keyword *const* indicates that a variable can be defined only once and the value must not be redefined.
- A variable may optionally be initialized by a value or an expression, which evaluates to a value of the given type.
- Variable declarations end by a semicolon.

Parameters of VIL build scripts are declared akin to variables, but without an initial value.

Example:

```
Integer numberOfCompilerProcesses = 4;
sequenceOf(Project) sources;
sequenceOf(Project, DecisionVariable) mapping;
```

Variables may be referred in Strings such as path patterns. A variable reference is stated as `$variableName`. Even entire VIL expressions (see Section 3.3) including variables may be given in Strings in the form `${expression}`. When applying the respective String, variable, and expression references are substituted with their actual value.

3.1.7 Externally Defined Values of Global Variables

Global variables or constants are defined as part of a VIL script. The value of a global variable or constant may be specified by an external source, e.g., to customize the build script according to the build environment (similar to properties in ANT [9]). For externally defined values of variables, initial values are not needed, in particular also not for constants. Externally specified values are subject to automated type conversion and variable reference or VIL expression substitution based on the VIL script arguments. Multiple external property files are processed in sequence so that the variable values defined by external files listed before are overwritten (accidental constant redefinition will lead to an execution error).

Syntax:

```
// Loading the values of global variables
load properties "path";
```

Description of Syntax: Loading values from an external file consists of the following elements:

- The keywords ***load properties*** indicates that the values of global variables shall be loaded from an external file. Multiple load statements may be given in a VIL script directly after the version statement.
- The path points to the file containing the initial values of the variables. Relative paths are interpreted relative to the target project of the VIL script. The file must be given in Java properties format, i.e., each line specifies the value of a specific variable in the following form
variableName = value
- Loading values from an external file ends by a semicolon.

Example:

```
load properties "globalVariables.properties";
```

3.1.8 Rules

Build rules are used in VIL to specify individual production strategies, reusable build steps to be used within production strategies or, as the main entry point into the build process. Akin to *make* [8], VIL-rules may have preconditions, which must be fulfilled in order to enable the rule. However, VIL-rules may also explicitly define postconditions, which guard the result of the rule execution.

VIL rules may have **parameters** in order to parameterize the specified variability instantiation. These parameters must either be bound by the calling rule or, in case of the main entry rule, by the VIL build script itself.

Preconditions may be given in terms of path-patterns, an individual artefact, an artefact collection or rule calls. While an arbitrary number of rule calls may be given as precondition, at most one path pattern, artefact or artefact collection may be given as first precondition⁴.

- A path pattern follows the (pattern) rules of ANT path specifications already described in Section 3.1.5.2. For example, "\$target/bin/**/*.class" requires the existence of at least one Java bytecode file in the bin folder of \$target (assuming that \$target refers to the target project). Used as a rule precondition, a path pattern requires that the matching file artefacts exist and are up-to-date (akin to Make rule preconditions [8] but with extended pattern matching capabilities).
- An artefact (collection) is given in terms of a variable or a VIL expression evaluating to exactly one artefact (collection) instance. In a precondition, the denoted artefact(s) must exist and be up-to-date.
- A rule call (rule name with argument list) represents an explicit rule dependency and must be executed successfully in case that the

⁴ Future work on VIL may relax this condition and even extend the current file path notation to more generic artifact model path expressions also involving fragment artefacts etc.

preconditions of the stated rule are valid. The execution results of a rule call become available as an implicit variable in the rule body under the name of the called rule.

The optional **rule postcondition** is given in terms of a path pattern, an individual artefact or an artefact collection⁴. Postconditions are evaluated if the preconditions are met and the body of the rule is executed successfully. A rule completes successfully, if also the (optional) postcondition is met.

Rules may explicitly depend on each other in terms of the rule calls described above. Further, **implicit rule dependencies** are expressed via the first (non-rule call) pre- or postcondition (akin to *make* rules [8]). If a path matching precondition⁴ for rule r_0 is not fulfilled, the VIL build language execution environment will aim at fulfilling the precondition by (recursively) searching for rules r_i with a postcondition indicating that the successful execution rule r_i contributes to the unmet precondition of rule r_0 . Ultimately, the possibly contributing rules r_i are executed (including their implicit rule dependencies) and the precondition of rule r_0 is checked again, and, on fulfilment, also r_0 is executed. If finally the precondition of r_0 is not fulfilled, r_0 is not considered for execution.

The rule body specifies the individual steps to be executed if the preconditions are met. A rule body may contain variable declarations, (assignment) expressions, explicit rule calls (not relevant as preconditions), instantiator calls, execution of system commands, or iterated execution of the previous elements. We will first describe the syntax of rules and describe then the individual statements available for specifying rule bodies.

If no path matching precondition is given, the rule body is executed once. If a path matching precondition is present, one or multiple artefacts may match that precondition and for each of these artefacts a corresponding output artefact may be required by the postcondition (if specified). Thus, the rule body is executed iteratively over all matching precondition artefacts. In order to address the actual artefact to be processed as well as its expected resulting artefact, the implicit variables `LHS` (in case of a matching precondition) and `RHS` (in case of a matching postcondition) will be made available to the loop body.

All rules return implicitly their execution results consisting of two sequences,

- `result` containing the immediately modified artefacts by that particular rule.
- `allResults` containing the modified artefacts by all dependent rule calls.

Further, the artefacts modified by executed rules are successively collected in the implicit global collection variable `OTHERPROJECTS`.

Syntax:

```
protected name (parameterList) = postcondition :  
    preconditions { // LHS/RHS may be available  
//variable declarations  
//rule, instantiator, artifact or system calls  
//iterated execution  
}
```

Description of Syntax: A rule declaration consists of the following elements:

- The optional keyword **protected** prevents that this rule is visible from outside so that such rules cannot be used as an entry point (for example, in ANT [9] this is expressed by a target name starting with the minus character). This does not affect the internal accessibility of rules via imports and rule call.
- The *name* allows identifying the rule for explicit rule calls or for script extension.
- The *parameterList* specifies explicit parameters which may be used as arguments for precondition rule calls as well as within the rule body. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameter must either be bound by the calling rule or, in case of the main entry rule, by the VIL build script itself (via identical names and assignable values in both, the template and the main sub-template).
- The first three parts may be omitted in case of anonymous rules which are only executed due to implicit dependencies and not available to explicit rule calls.
- The optional postcondition specifies the expected outcome of the rule execution. A postcondition may be a path match, an artefact or an artefact collection. In case of a path match, the implicit variable RHS will be made available to the rule body.
- The optional preconditions specify whether the rule is considered for execution. The first precondition (made available as implicit variable LHS to the rule body) may be a path match, an artefact or an artefact collection. The following preconditions may be explicit rule calls. The execution results of the preconditions will be made available to the rule body in terms of implicit variables with names of the called rules and the rule return type described above.
- The rule body is specified within the following curly brackets.

Example:

```
produceGenericCopy(FileArtifact x, FileArtifact y) = y : x
{
    x.copy(y);
}

compileGoal() = "$target/bin/*.class" : "$source/*.java"
{
    javac(RHS, LHS);
}
```

The rule body specifies the individual steps to be performed in order to fulfil the rule postcondition (if stated). A rule body may contain variable declarations, (assignment) expressions, explicit rule calls, instantiator calls, execution of system commands or iterated execution of these elements. The statements (ended by a semicolon or a statement block) given in a rule body are executed in the given sequence. We will discuss these individual elements in the following subsections.

3.1.8.1 Variable Declarations

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within nested blocks. Basically, a variable declaration within a rule body follows the same syntax as global variable declarations discussed in Section 3.1.6.

3.1.8.2 Expressions

Expressions such as value calculations or execution of artefact operations may be used within a rule body as a guard expression or as a variable assignment. Please note that we will detail the VIL expression language in Section 3.3, as the expression language is common to both, the VIL build language and the VIL template language.

- Guard expressions constrain the execution of the remaining statements in a rule body, i.e., the expression must be evaluated successfully in order to continue the execution of the rule.
- In a variable assignment, the expression on the right hand side of the assignment operator “=” must be evaluated successfully in order to assign the evaluation result of the right side to the variable specified on the left side.

3.1.8.3 Calls

A call leads to the execution of another build language rule, an instantiator or an artefact operation. We will discuss three types of calls in this section, as they are represented by the same syntax. However, the most extreme call of a (blackbox) instantiator, namely the execution of an operating system command (including operating system scripts) follows a slightly different syntax. We will discuss operating system commands in Section 3.1.8.4.

The syntax of rule calls, instantiators or artefact operations looks as follows:

operationName(argumentList)

whereby arguments are expressions separated by commas. Calls may return values of different type.

Rule Calls

An explicit rule call is stated in terms of the name of the rule and the arguments matching the parameter list of the target rule. A rule call leads to the execution of a build language rule defined in the same script, one of the extended scripts or an imported script. As rules with the same signature consisting of name and parameter list are shadowed by the extension, rules in extended scripts may explicitly be called by

`super.operationName(argumentList)`

Instantiator Calls

Basically, the VIL build language aims at defining the production flow for instantiating generic artefacts for a software product line. In contrast, the VIL template language aims at specifying the individual actions to instantiate an individual (generic) artefact. Further instantiators may be given in terms of (wrapping) Java classes in order to make programming language compilers, linkers, or legacy instantiators available. Such instantiators may provide information about their execution, in particular the created artefacts.

In the VIL build language, all these types of instantiators are mapped transparently to one kind of statement, the instantiator call:

`operationName(argumentList)`

Basically, an instantiator call looks similar to a rule call, i.e., a name with a parameter list, but it (typically) returns a collection of artefacts (or even nothing in case of wrapped blackbox instantiators). Instantiators may be rather generic (such as the built-in instantiator for the VIL template language) and may offer to pass an arbitrary number of arguments (e.g., those defined by a VIL template. Therefore, depending on the instantiator, named arguments (`parameterName = valueExpression`) may pass arbitrary VIL instances to an instantiator in a generic way.

We will detail the built-in instantiators in Section 3.4.6. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize an instantiator.

Artefact Operation Calls

Artefact operations provide information on an individual artefact, its fragments or even enable the manipulation of artefacts. Basically, an artefact operation is executed on a variable or expression, which evaluates to an artefact type. An artefact operation can be expressed (akin to IVML and OCL) in two different ways, using the artefact as first argument

`operationName(artefact, argumentList)`

or in object-oriented style

`artefact.operationName(argumentList)`

Basically, a `String` can be automatically converted into a `Path` or an `Artifact`. Similarly, a `Path` can be transparently converted into an `Artifact`. However, in some cases, also an explicit creation of an artefact of a certain type may be required. Typically, the individual artefact types support the following constructors

```
new ArtefactType(String)
```

for obtaining a specific artefact specified by its path. Please note that artefacts are associated with creation rules detailed in Section 3.4.5. Basically, file artefacts (regardless of whether they physically exist or only the path is known) are polymorphically determined according to their file name extension, e.g., a file with extension `xml` is considered to be a `XmlFileArtifact`. Further, content-specific rules may apply depending on the specific artefact type. If no such rule applies, a basic `FileArtifact` is created as the default fallback. Thus, the underlying mechanisms of the VIL artefact model will check whether the creation of that instance (regardless of whether the underlying file exists or not) is actually possible or not. If the creation fails, also the containing rule will fail. The constructor

```
new ArtefactType( )
```

allows to obtain a temporary artefact. Unless not renamed, this artefact will be automatically deleted after terminating the execution of the VIL script.

The modifications to a VIL artefact instance will automatically be synchronized with the underlying artefact upon the end of the lifetime of the related variable, e.g., when the execution of the containing scope of a local variable ends.

We will detail the built-in artefact operations in Section 3.4.5. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize own artefact types and related operations.

Operation Resolution

While determining the applicable rules, instantiators, or artefact operations, the VIL type system considers in the following sequence

- (1) Exact match of argument types and parameter types.
- (2) Assignment compatible argument types and parameters.
- (3) Implicit conversions specified as part of the implementation of VIL types and artefacts, e.g., the implicit conversion of a `String` to a `Path` or a `String` to an `Artifact`. Details on the type system and the available conversions will be discussed in Section 3.3.

The operation types discussed in this section will be resolved according to the sequence below:

- (1) Rule calls
- (2) Instantiator calls
- (3) Artefact, configuration type, and basic type operations

Further, the VIL runtime environment performs dynamic dispatch, i.e., the operation determined and bound at script parsing time will be reconsidered with respect to the actual types of parameters and the best matching operation will dynamically be

determined (similar to dynamic dispatch in Xtend [2]). This avoids the need for explicit type checking or large alternative decision blocks.

3.1.8.4 Operating System Commands

The VIL build language is also able to execute the most basic form of a blackbox instantiator, namely operating systems calls or scripts. However, the syntax for system calls differs slightly from the other call types discussed in Section 3.1.8.3 as operating system commands may require explicit path specifications.

```
execute identifier(argumentList)
```

whereby *identifier* must denote a variable which evaluates to a `String` or a `Path`. This enables that operating system calls can be composed at script execution time or determined using external values (see Section 3.1.7). However, the related command or script is executed, but the created artefacts are not tracked by the VIL execution environment.

3.1.8.5 Iterated Execution

Finally, all statements available in a rule body may explicitly be executed in iterative fashion, e.g., to apply a sequence of instantiator calls explicitly to a container of artefacts. Therefore, the VIL build language offers a dedicated loop statement. However, this statement called `map` in the VIL build language is different from typical programming language loops as it collects the result of its execution in terms of modified artefacts (similar to a rule).

Syntax:

```
map(names = expression) {  
  //variable declarations  
  //rule, instantiator, artefact or system calls  
  //iterated execution  
}
```

Description of Syntax: An iterated execution consists of the following elements:

- The keyword **map** followed by parenthesis defining the iterator variables.
- The *names* denoting the names of the variables used by the map expression iterate. The number, type and the contents of the iterator variables are implicitly defined by the related *expression*. Typically, the expression will lead to a container with one parameter type so that map will iterate over that collection using exactly one variable of the element type of the container. However, as we will discuss in Section 3.1.8.6, the `join` expression may return a multi-dimensional container, which then needs multiple iterator variables.
- The **map** consists of a block determining the statements to be executed in for an individual iteration. The *names* denoting the iterator variables shall be used within the block.

Example:

```
map(d : config.variables()) {  
    // operate on the iterator variable d of type  
    // DecisionVariable (see Section 3.4.4.6)  
}
```

3.1.8.6 Join Expression

One specific expression in the VIL build language is particularly intended to be used with the `map` iteration statement, namely the `join` operation. However, as `join` is an expression, it may be used as an usual expression, e.g., on the right hand side of a value assignment to a variable.

This operation is inspired by database joins, e.g., as usual in SQL. The `join` operation allows combining containers of different VIL types, in particular elements from the variability configuration with source or target artefacts. Depending on type of the specified expression types, the `join` operation returns a typed sequence containing the results.

Syntax:

```
join(name1:expression1, name2:expression2) with (expression)
```

Description of Syntax: A VIL join expression consists of the following elements:

- The keyword `join` followed by one parenthesis defines the containers to be joined and the related iterator variables (*name₁*, *name₂*).
- *name₁*, *name₂* denote variables used by the join expression iterate over the containers given in *expression₁* and *expression₂*. Without further restriction, the result will be a collection of pairs on the types parameterized by the types of *expression₁* and *expression₂*. The keyword `exclude` used before one of the names leads to a left- or right-sided join, thus restricting number of parameters of the resulting collection to one.
- The third *expression* specifies the join condition, i.e., an expression involving *name₁* and *name₂* to select the relevant results from the cross product of *name₁* and *name₂*, and to effectively reduce the size of the result.

Example:

```
// work on those decisions and artifacts where a certain  
// string composed from the decision name occurs in the  
// artifact (and may be substituted by an instantiator)  
map(d, a :  
    join(d:config.variables(), a:"$source/src/**/*.java")  
    with (a.text().matches("${" + d.name() + "}")) {  
        // operate on decision variable d and  
        // related artifact a  
    }
```

3.2 VIL Template Language

In this section, we describe the concepts and language elements of the VIL template language in detail. In contrast to the VIL build language, which aims at specifying the instantiation of all artefacts of a product line, the VIL template language aims at specifying the instantiation of a single artefact.

3.2.1 Reserved Keywords

In the VIL template language, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by those of the common VIL expression language given in Section 3.3.1.

- `@advice`
- `@indent`
- `const`
- `def`
- `default`
- `else`
- `extends`
- `extension`
- `for`
- `if`
- `import`
- `print`
- `switch`
- `template`
- `version`
- `with`

3.2.2 Template

The template (`template`) is the top-level structure in the VIL template language. This element is mandatory as it defines the frame for specifying how to instantiate a certain artefact. Please note that exactly one template must be given in a VIL template file.

The definition of a template requires a name, which acts for referring among VIL templates and a parameter list specifying the expected information from the calling VIL build script such as the actual configuration and the target artefact (fragment). Please note that these two arguments must be provided to all VIL template scripts.

Basically, VIL may refer to all visible configuration settings in a variability configuration, more precisely to those actual values of decision variables (and their underlying structure), which are frozen. In order to make this integration explicit, these decision variables may be directly referenced in the VIL template language by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. To support the domain engineer in

specifying valid templates, also the VIL template language provides the **advice** annotation (see also Section 3.1.2).

Optionally, a VIL template may extend another VIL template, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

The VIL template language particularly aims at supporting generative and manipulative instantiation of generic artefacts. Therefore, the VIL template language provides capabilities for easily specifying and generating contents. However, as usual in software development, also VIL templates shall be formatted properly. In order to distinguish between intended formatting and whitespaces that shall not occur in the target artefact (fragment), the VIL template language is able to take the actual indentation into account (as specified in the **indent** annotation). Taking the formatting of the templates into account avoids postprocessing of the results, e.g., by formatting mechanisms [2]. We will discuss the indentation processing of the VIL template language as part of the content statements in Section 3.2.8.6.

Akin to programming languages, VIL templates may contain (global) variable declarations as well as sub-templates (similar to methods in object-oriented programming languages or functions in the structured programming paradigm).

Syntax:

```
//imports
//functional extensions
@advice(ivmlProjectName)
@indent(indentationSpec)
template name (parameterList) extends name1 {
    //optional version specification
    //variable definitions
    //sub-template declarations
}
```

Description of syntax: The definition of a VIL template consists of the following elements:

- Optionally, imported templates or functional extensions by Java classes are listed first.
- Optional advices declaring the underlying variability models. This annotation is similar to VIL (see Section 3.1.2).
- An optional indentation annotation enabling the VIL template execution to take the actual indentation into account when processing content statements. We will detail the use of the indentation annotation in Section 3.2.8.6 along with the content statement, which actually considers indentation information.
- The keyword **template** defines that the following identifier *name* defines a new artefact instantiation template.

- The parameter list denotes the arguments a VIL template requires when being executed. Basically, a VIL-template receives the underlying variability configuration and the target artefact as parameters⁵. The arguments are subject to dynamic dispatch, i.e., either the most generic type `Artifact` may be used for the target artefact or a more specific type can be used. In the latter case, the instantiator statement in the VIL build script must also pass in a type-compliant artefact instance. Additional parameters may be defined which then must be stated in the calling VIL build script as named arguments.
- A VIL template may optionally extend an existing (imported) VIL template. This is expressed by **extends** *name₁*, whereby *name₁* denotes the name of the extending script.
- The optional version specification, variable declarations and sub-templates are then stated within the curly brackets.

Example:

```
@advice(YMS)
template DbCreator (Configuration config,
    Artifact target){
    /* Go on with description of the artefact instantiation
       starting with a main sub-template and possibly further
       (imported) sub-templates */
}
```

3.2.3 Version

Akin to IMVL and the VIL build language, also the VIL template language can be tagged with an explicit version number in order to support evolution. The syntax for the version declaration is identical to the VIL build language as discussed in Section 3.1.3.

3.2.4 Imports

The description of the instantiation of a certain artefact type may be defined in a single VIL template (possibly including sub-templates) or may be composed from reusable sub-templates specified in other (existing) build scripts. Therefore, VIL templates may be imported. In order to support also the evolution of product line build specifications, also the VIL template language allows the specification of version-restricted imports. Imports make the sub-templates defined in the specified build file accessible to the importing template. The syntax of imports in VIL templates is identical to imports in the VIL build language as discussed in Section 3.1.4.

⁵ Due to the current implementation, the configuration parameter must be named "config" and the target artefact "target".

3.2.5 Functional Extension

Sometimes, it is necessary to realize specific supporting functions such as calculations in terms of a programming language rather than in the template language itself. Therefore, similar to Xtend [2], the VIL template language enables external functions in terms of static Java methods, to call these methods from the template language and to use the results in VIL templates. Basically, the realizing classes are declared in VIL as extension and containing static methods are made available as they would be VIL operations. However, methods with already known signatures will not be redefined.

Syntax:

```
extension name;
```

Description of Syntax: A functional extension in the VIL template language consists of the following elements:

- The keyword **extension** followed by a qualified Java *name* denoting the class to be considered. The referred class must be available to VIL through class loading. Contained static methods will be considered as extension methods for the VIL template language. Please note that the implementing method shall use only primitive Java types or (the implementation classes of the) VIL types discussed in Section 3.3.
- An extension declaration ends with a semicolon.

Example:

```
extension java.lang.System;
```

3.2.6 Types

Basically, the VIL template language is a statically typed language with some convenience in terms of postponed type checking at runtime akin to the VIL build language. Thus, the VIL template language provides a set of formal types available for variable declarations or parameter lists. VIL template language and VIL build language rely on the same type system and, thus, the VIL template language provides the same types as discussed in Section 3.1.5.

3.2.7 Variables

A variable provides named access to a value of a certain type similar to variables in programming languages. The semantic of variables as well as the syntax for declaring and using them in the VIL template language is identical to the VIL build language as discussed in Section 3.1.6 (except for the capability of defining variable values in an external file which is not available in the VIL template language).

Similar to the VIL build language, variables may be referred in Strings such as paths or content statements. A variable reference looks like `$variableName`. Even entire VIL expressions (see Section 3.3) including variables may be given in the form

`${expression}`. When applying the respective element, variable and expression references are substituted by their actual value.

3.2.8 Sub-Templates (defs)

The actual instantiation of an artefact is given in terms of sub-templates (called `def` in the concrete syntax), i.e., named functional units with parameters and return types. One specific sub-template (usually called `main`) acts as the entry point into artefact instantiation. Akin to the VIL build language, it receives the parameters of the containing template as arguments.

The body of a sub-template specifies the individual steps to be executed for realizing the instantiation. Such a body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). We will first describe the syntax of templates and discuss then the statements available in sub-template bodies.

Syntax:

```
def name (parameterList) {  
  //variable declarations  
  //alternative, switch-case, loop  
  //content statements  
}  
  
def Type name (parameterList) {  
  //variable declarations  
  //alternative, switch-case, loop  
  //content statements  
}
```

Description of Syntax: A sub-template declaration consists of the following elements:

- The keyword **def** indicates the definition of a sub-template.
- By default, the VIL template language aims at inferring the return type of a sub-template from the rule body. In the extreme case, individual statements may produce a rather generic value of type `AnyType`, which enables the use of the value without type checking at template parsing time and type checking at runtime. However, in some situations the template developer may explicitly want to do strict type checking at template parsing time. This is enabled by specifying the optional return type for a sub-template.
- The *name* allows identifying the sub-template for calls, template extension or as `main` entry point.
- The *parameterList* specifies the parameters of a sub-template in order to parameterize the instantiation operations subsumed by the

respective sub-template. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameters must either be bound by the calling sub-template or, in case of the main entry rule, by the VIL template itself (via identical names and assignable types, the template and the main sub-template).

- The rule body is specified within the following curly brackets.

Example:

```
def main(Configuration config, Artifact target) {  
    // define artifact instantiation  
}  
  
def String valueMapping (DecisionVariable var) {  
    // map the value of var to a String  
    // explicit type checking is enforced  
}
```

The sub-template body specifies the individual steps needed to instantiate an artefact. Such a rule body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). The statements (ended by a semicolon or a statement block) given in a sub-template body are executed in the given sequence. We will discuss these individual statement types in the following subsections.

The last statement executed in a sub-template body implicitly determines the return value of a sub-template.

3.2.8.1 Variable Declaration

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within statement-blocks such as loops or alternatives. Basically, a variable declaration within a sub-template body follows the same syntax as global variable declarations discussed in Section 3.2.7.

3.2.8.2 Expression Statement

Expressions such as value calculations or execution of artefact operations may be used within a sub-template body as a guard expression or as a variable assignment. Please note that we will detail the VIL expression language in Section 3.3, as the expression language is common to both, the VIL build language and the VIL template language. Thus, guard expressions and variable assignments as discussed in Section 3.1.8.2 are similarly available in the VIL template language. Further, similar call types as well as their (resolution) semantic as discussed in Section 3.1.8.3 are available in the VIL template language (of course, rule calls are replaced by template calls and operating system calls by calls to functional extensions). Template calls may be recursive. In the VIL template language, the resolution sequence is

- 1) Template calls
- 2) Artefact, configuration type and basic type operations

3) Functional extension calls

3.2.8.3 Alternative

In the VIL template language, alternatives allow choosing among different ways of instantiating an artifact, i.e., upon evaluating a condition the appropriate alternative to execute is determined.

The (return) type of an alternative is either the common type of all alternatives or `AnyType`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
if (expression) ifStatement  
if (expression) ifStatement else thenStatement
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **if** indicates the beginning of an alternative statement.
- The *expression* given in parenthesis determines whether the if-part (condition is evaluated to true) or the else-part (condition is evaluated to false) is executed. In particular, if the *expression* cannot be evaluated, evaluation will terminate the evaluation of the sub-template block before evaluating the alternatives.
- The *ifStatement* (or statement block enclosed in curly braces) is being executed when the *expression* is evaluated to true. A single statement must be terminated by a semicolon.
- The **else** part is optional, i.e., if else is used in an alternative, a following *thenStatement* is required. As usual, a dangling else is bound to the innermost alternative.
- The *thenStatement* (or statement block enclosed in curly braces) is executed if the *expression* is evaluated to false. Again, a single statement must be terminated by a semicolon.

Example:

```
if (config.variables().size() > 0) {  
    // work on config  
} else {  
    // produce an empty artefact  
}
```

3.2.8.4 Switch

The switch statement in the VIL template language is for (dynamically) mapping configuration elements to artefact elements rather than for influencing the control flow (as it is the case for the alternative statement). However, in case of larger mappings with (more or less) static content, we suggest using a map variable (see Section 3.1.5.4).

The (return) type of an alternative is either the common type of all cases or `AnyType`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
switch (expression) {  
    expression1 : expression2,  
    expression3 : expression4  
}
```

```
switch (expression) {  
    expression1 : expression2,  
    default : expression5  
}
```

```
switch (expression) {  
    expression1 : expression2,  
    expression3 : expression4,  
    default : expression5  
}
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **switch** indicates the beginning of a switch statement. It is followed by an *expression* to switch over and the individual cases in a block of curly brackets.
- A case consists of an expression (*expression*₁ or *expression*₃ above) to be matched against expression. If an individual match succeeds, the related value expression will be evaluated and determines the (result) value of the switch statement. The implicit variable `VALUE` may be used within the value expression in order to refer to the evaluated value of *expression*.
- Optionally, a **default** case can be given which is considered if none of the previous cases matches. In that case, the expression stated behind the default will be evaluated and determines the (result) value of the switch statement.

Example:

```
switch (var.name()) {  
    "forkNumber" : VALUE + var.intValue() - 1,  
    "cpuNumber" : var.stringValue()  
    //go on with further cases and a default value  
    //if required  
}
```

3.2.8.5 Loop

The for-statement in VIL enables the defined repetition of statements. Basically, it is rather similar to an iterator-loop in Java.

Syntax:

```
for (Type var : expression) statement
```

```
for (Type var : expression, expression1) statement
```

Description of Syntax: A loop statement consists of the following elements:

- The keyword **for** indicates the beginning of a for-loop statement. It is followed by a parenthesis defining the loop iterator, i.e., a variable to which successively all values of the *expression* are assigned. Therefore, *expression* must either evaluate to a set or a sequence.
- The statement (or statement block enclosed in curly braces) is then executed for each element in the collection specified by *expression* while the iterator *var* is successively assigned to each individual value in the collection.
- An optional separator expression *expression*₁ which is evaluated and emitted (without line end) at the end of each iteration if further iterations will happen. Such a separator expression simplifies generating value lists.

Example:

```
for (DecisionVariable var: config.variables()) {  
    //operate on var  
}  
  
for (DecisionVariable var: config.variables(), ", ") {  
    //print var (the separator will be added if needed)  
}
```

3.2.8.6 Content

The content statement is used to generate the content of the target artefact. Basically, all characters given in a String (enclosed in a pair of apostrophs or quotes including appropriate Java escapes and line breaks) are emitted as output to the result artefact. Content statements executed in the course of template evaluation according to the control flow make up the entire content of the target artefact (fragment). A content statement may consist of multiple lines as part of the content. Thereby, variable references or IVML expressions are substituted as described for variables in Section 3.2.7.

Without further consideration, also the indentation whitespaces for pretty-printing a VIL-template will be taken over into the resulting artefact. In order to provide more control about the formatting, the annotation **@indent** allows specifying the number

of whitespaces used as one indentation step (value `indentation`), the number of whitespaces to be considered in tabulator emulation (value `tabEmulation`) and also how many additional whitespaces (value `additional`) are used to indent the content statement, i.e., whether the following lines after the lead in character are further subject to indentation or not. Further whitespaces in the content are considered as formatting of the content itself and are taken over into the artefact. In addition, an optional numerical value can be specified at each content statement in order to programmatically indent the configuration by the given number of whitespaces.

Syntax:

```
"text"  
  
print "text"  
  
"text" | expression;  
  
'text' | expression;
```

Description of Syntax: A content statement consists of the following elements:

- The optional keyword **print**, which indicates that no line end shall be emitted at the end of the content. If absent, implicitly a print-line is performed. The print form of the content statement is helpful in combination with the separator expression in loops.
- The lead in / lead out marker (apostrophe or quote) indicates the content statement and marks the actual content. Two forms are used to enable the use of the opposite character in content. A content statement may cover multiple lines of content.
- An optional indentation expression can be indicated by the pipe symbol and followed by a numerical *expression*. The numerical *expression* determines the amount of whitespaces to be used as mandatory indentation prefix for each individual line of the actual content statement. Only if the indentation expression (may be a constant or a true expression) is specified, a semicolon is required.

Example:

```
'CREATE DATABASE ${var.name()}'  
'CREATE TABLE data' | 4;
```

3.3 VIL Expression Language

In this section, we will define the syntax and the semantics of the VIL expression language, which is common to the VIL build language and the VIL template language. Similar to IVML, expressions in the VIL languages are inspired by OCL. Thus, most of the content in this section is taken from OCL [6] or the IVML language specification [3, 7] and adjusted to the need, the notational conventions, and the semantics of the VIL languages, in particular also regarding the syntax of the iterators and the absence of quantors in VIL.

3.3.1 Reserved Keywords

Keywords in the VIL expression language are reserved words. That means that the keywords must not occur anywhere in an expression as the name of a rule, a template or a variable. The list of keywords is shown below:

- **and**
- **false**
- **new**
- **not**
- **or**
- **sequenceOf**
- **setOf**
- **super**
- **true**
- **xor**

In order to increase reuse among the VIL languages, the VIL expression language also provides the definition of common language concepts such as variable declarations and parameter lists. The related keywords were already listed in Section 3.1.1 and 3.2.1, respectively.

3.3.2 Prefix operators

The VIL expression language defines two prefix operators, the unary

- Boolean negation '**not**' and its alias '!'.
Note: The keyword **not** is used in the original OCL specification, but it is not a reserved keyword in VIL.
- Numerical negation '-' which changes the sign of a Real or an Integer.

3.3.3 Infix operators

Similar to OCL, in VIL the use of infix operators is allowed. The operators '+', '-', '*', '/', '<', '>', '<=>', '<=' '>=' are used as infix operators. If a type defines one of those operators with the correct signature, they will be used as infix operators. The expression:

$a + b$

is conceptually equal to the expression:

$a . + (b)$

that is, invoking the "+" operation on *a* (the *operand*) with *b* as the parameter to the operation. The infix operators defined for a type must have exactly one parameter.

For the infix operators '<,' '>,' '<=,' '>=,' '<>,' 'and,' 'or,' 'xor', the return type is Boolean.

Please note that, while using infix operators, in VIL Integer is a subclass of Real. Thus, for each parameter of type Real, you can use Integer as the actual parameter. However, the return type will always be Real. We will detail the operations on basic types in Section 3.4.2.

3.3.4 Precedence rules

The precedence order for the operations, starting with highest precedence, in IVML is:

- dot operations: '.' (for operation calls in object-oriented style)
- unary 'not', !(alias for not) and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- '<,' '>,' '<=,' '>='
- '==' (equality), '<>,' '!=' (alias for '<>')
- 'and,' 'or' and 'xor'

'(' and ')' can be used to change precedence.

3.3.5 Datatypes

All artefacts defined by the extensible VIL artefact model as well as the various built-in types are available to the expression language and may be used in expressions. IVML elements are mapped into VIL via IVML qualified names. Figure 1 illustrates the VIL type hierarchy (not detailing the IVML integration through the configuration types). Below, we discuss the use of datatypes.

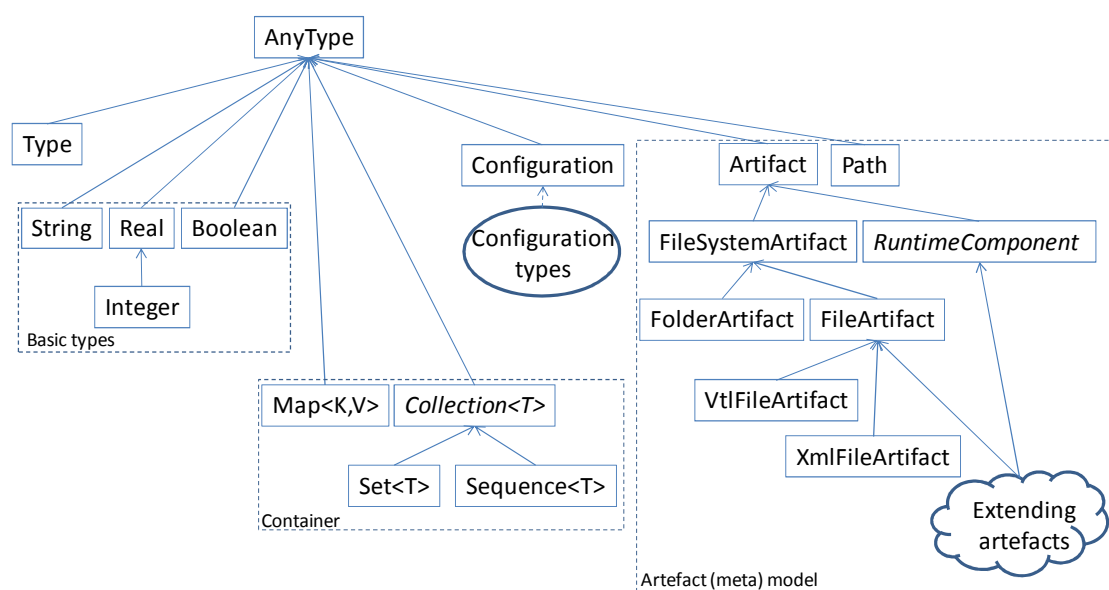


Figure 1: Overview of the VIL type system

3.3.6 Type conformance

Type conformance in IVML constraints is inspired by OCL (cf. OCL section 7.4.5):

- `AnyType` is the common superclass of all types. All types comply with `AnyType`. However, `AnyType` is not directly available to the user and used internally to denote expressions of unknown type to be resolved and checked while execution time of the specifying script.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies or if no type parameters are given (deferred to script evaluation time).

3.3.7 Side effects

In contrast to OCL, some constraint expressions in IVML may lead to side effects, in particular to modifications of artefacts and artefact fragments.

3.3.8 Undefined values

Basically, variables and IVML qualified names, i.e., links into a variability model, may be undefined. During evaluation of expressions at script or template execution time, undefined (sub-)expressions are ignored and do not lead to failing of rules or sub-templates.

3.3.9 Collection operations

The VIL artefact model and the IVML integration into VIL in terms of the `Configuration` type define many operations returning collections. Operating with collections is specifically meant to enable a flexible and powerful way of accessing information and for deriving artefacts.

Practically, collections in VIL are sets, sequences, or maps as introduced in Section 3.1.5.4. Collections are a basis for iterations in the VIL template language (Section 3.2.8.5) as well as for joins and map iterations in the VIL build language (Section 3.1.8.5). Therefore, we defined a set of basic operations on the artefact types and the `Configuration` providing convenient access through selection and filtering. However, with an increasing amount of information provided by the accessible types, more and more collection access operations and related changes to the VIL types will be needed. However, we refrained from a specific syntax for these operations as in IVML and rely on implicit iterator variables similar to other implicit variables in the VIL languages. One example is the **`select`** operation, which returns all

elements of a collection complying to a certain Boolean selection expression. An example for an expression with a generic iterator is

```
configuration.variables().select(VAR.name().length() = 10)
```

Which returns those decision variables with a name of length 10 (VAR is the generic iterator variable implicitly provided by the VIL language for such specific collection operations).

3.4 Built-in operations

Similar to OCL and IVML, all operations in VIL are defined on individual types and can be accessed using the “.” operator, such as `set.size()`. However, this is also true for the equality, relational, and mathematical operators but they are typically given in alternative infix notation, i.e., `1 + 1` instead of `1.+(1)` as stated in Section 3.3.3. Further, the unary negation is typically stated as prefix operator. Due to the VIL artefact model, the integration with IVML and the specific purpose of variability instantiation, the VIL types define a different set of operations than OCL/IVML⁶.

In this section, we denote the actual type on which an individual operation is defined as the *operand* of the operation (called *self* in OCL). The parameters of an operation are given in parenthesis. Further, we use in this section the Type-first notation to describe the signatures of the operation.

Please note that those operations starting with “get” (Java-getters) are also available with their short name without “get” in order to simplify script and template creation, e.g., the `getName()` operation is also available as its aliased operation `name()`. We will make this explicit by listing both operation signatures.

3.4.1 Internal Types

3.4.1.1 AnyType

`AnyType` is the most common type in the VIL type system. All types in VIL are type compliant to `AnyType`. In particular, `AnyType` is used as type if the actual type is unknown at parsing time and shall be determined dynamically at runtime. Therefore, `AnyType` can be assigned to variables of any type (but no specific operations can be executed on `AnyType`).

3.4.1.2 Type

`Type` represents type expressions themselves and enables the type-generic selection type-compliant elements from collections.

3.4.2 Basic Types

In this section, we detail the operations for the basic VIL types.

⁶ An extended set of operations will be defined by future extensions of VIL.

3.4.2.1 Real

The basic type `Real` represents the mathematical concept of real numbers following the Java range restrictions for double values. Note that `Integer` can automatically be converted to `Real`.

- **Real + (Real r)**
The value of the addition of *self* and the *operand*.
- **Real - (Real r)**
The value of the subtraction of *r* from the *operand*.
- **Real * (Real r)**
The value of the multiplication of the *operand* and *r*.
- **Real - ()**
The negative value of the *operand*.
- **Real / (Real r)**
The value of the *operand* divided by *r*. Leads to an evaluation error if *r* is equal to zero.
- **Boolean < (Real r)**
True if the *operand* is less than *r*.
- **Boolean > (Real r)**
True if the *operand* is greater than *r*.
- **Boolean <= (Real r)**
True if the *operand* is less than or equal to *r*.
- **Boolean >= (Real r)**
True if the *operand* is the same as *r*.
- **Boolean = (Real r)**
True if the *operand* is the same as *r*.
- **Real abs()**
The absolute value of the *operand*.
- **Integer floor ()**
The largest integer that is less than or equal to the *operand*.
- **Integer ceil()**
The closest integer value that is greater or equal to the *operand*.
- **Integer round()**
The integer that is closest to *the operand*. When there are two such integers, the largest one.

3.4.2.2 Integer

The standard type `Integer` represents the mathematical concept of integer numbers following the Java range restrictions for integer values. Note that `Integer` is a subclass of `Real`.

- **Integer + (Integer i)**
The value of the addition of the *operand* and *i*.
- **Integer - (Integer i)**
The value of the subtraction of *i* from the *operand*.
- **Integer * (Integer i)**
The value of the multiplication of the *operand* and *i*.

- **Real / (Integer i)**
The value of the *operand* divided by *i*. Leads to an evaluation error if *i* is equal to zero.
- **Boolean < (Integer i)**
True if the *operand* is less than *i*.
- **Boolean > (Integer i)**
True if the *operand* is greater than *i*.
- **Boolean <= (Integer i)**
True if the *operand* is less than or equal to *i*.
- **Boolean >= (Integer i)**
True if the *operand* is greater than or equal to *i*.
- **Boolean == (Integer i)**
True if the *operand* is the same as *i*.
- **Integer - ()**
The negative value of the *operand*.
- **Integer abs()**
The absolute value of the *operand*.

Conversions: `Integer` values can automatically be converted to `Real` values.

3.4.2.3 Boolean

The basic type `Boolean` represents the common true/false values.

- **Boolean == (Boolean a)**
True if the *operand* is the same as *a*.
- **Boolean or (Boolean b)**
True if either *self* or *b* is true.
- **Boolean xor (Boolean b)**
True if either *self* or *b* is true, but not both.
- **Boolean and (Boolean b)**
True if both *b1* and *b* are true.
- **Boolean not ()**
True if *self* is false and vice versa.
- **Boolean ! ()**
True if *self* is false and vice versa.

3.4.2.4 String

The standard type `String` represents strings, which can be ASCII.

- **Integer length ()**
The number of characters in the *operand*.
- **String + (String s)**
The concatenation of the *operand* and *s*.
- **String + (Path p)**
The concatenation of the *operand* and the string representation of path *p*.
- **String substring(Integer start, Integer end)**

Returns the substring of the *operand* from *start* (inclusive) to *end* (exclusive). *operand* is returned in case of any problem, e.g., positions exceeding the valid index range.

- **Boolean == (String s)**
True if the *operand* is the same as *s*.
- **Boolean matches (String r)**
Returns whether the *operand* matches the regular expression *r*. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether `mail` is a valid e-mail-address:

```
mail.matches([\w]*@[\w]*.[\w]*);
```
- **Integer toInteger ()**
Converts the *operand* to an Integer value.
- **Real toReal ()**
Converts the *operand* to a Real value.

3.4.3 Container Types

This section defines the operations of the collection types. VIL defines one abstract collection type `Collection` and two specific collections, namely `Set` and `Sequence`. All collection types are parameterized by one parameter. Below, 'T' will denote the parameter for the collection types. A concrete collection type is created by substituting a type for the parameter T. So a collection of integers is referred in VIL by `setOf(Integer)`.

In addition, VIL defines the type `Map`, an associative container, which allows to relate keys to values.

3.4.3.1 Collection

`Collection` is the abstract super type of all collections in VIL.

- **Integer size ()**
The number of elements in the collection *operand*.
- **Boolean includes (T object)**
True if *object* is an element of *operand*, false otherwise.
- **Boolean excludes (T object)**
True if *object* is not an element of *operand*, false otherwise.
- **Integer count (T object)**
The number of times that *object* occurs in the collection *operand*.
- **Boolean ==(Collection<Type> c)**
Returns whether *operand* contains the same elements than *c*, for ordered collections such as `Sequence` also whether the elements are given in the same sequence.
- **Boolean equals(Collection<Type> c)**
Returns whether *operand* contains the same elements than *c*, for ordered collections such as `Sequence` also whether the elements are given in the same sequence.
- **Boolean isEmpty ()**
Is the *operand* the empty collection?

- **Boolean isEmpty ()**
Is the *operand* not the empty collection?

3.4.3.2 Set

The type `Set` represents the mathematical concept of a set. It contains elements without duplicates. `Set` inherits the operations from `Collection`.

- **Sequence<T> toSequence ()**
Turns *operand* into a sequence.
- **Set<Type> selectByType (Type t)**
Returns all those elements of *operand* that are type compliant to *t*.
- **Set<T> excluding (Collection<T> s)**
Returns a subset of *operand*, which does not include the elements in *s*.
- **Set<T> select (Expression e)**
Returns the elements in *operand*, which comply with the iterator expression *e* (via the implicit iterator variable `VAR`).

3.4.3.3 Sequence

A `Sequence` is a `Collection` in which the elements are ordered. An element may be part of a `Sequence` more than once. `Sequence` inherits the operations from `Collection`.

- **T get (Integer index)**
Returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than `size()`.
- **T [] (Integer index)**
The `[]`-operator returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than `size()`.
- **Boolean ==(Collection<Type> c)**
Returns whether *operand* contains the same elements in the same sequence than *c*.
- **Boolean equals(Collection<Type> c)**
Returns whether *operand* contains the same elements in the same sequence than *c*.
- **Set<T> toSet ()**
Turns *operand* into a set (excluding duplicates).
- **Sequence<T> selectByType (Type t)**
Returns all those elements of *operand* that are type compliant to *t*.
- **Sequence <T> excluding (Collection<T> s)**
Returns a subset of *operand*, which does not include the elements in *s*.
- **Sequence <T> select (Expression e)**
Returns the elements in *operand* which comply with the iterator expression *e* (via the implicit iterator variable `VAR`).

3.4.3.4 Map

The `Map` type represents an associative container, which is parameterized over the type of keys `K` and the type of values `V`.

- **V get (K key)**
Returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than *size()*.
- **V [] (K key)**
The []-operator returns the value assigned to the given *key* in *operand*.

Sequences which contain sequences with exactly two entry types matching the types of `Map` can be converted automatically into a `Map` instance.

3.4.4 Configuration Types

Configuration types realize the integration with IVML. As the VIL languages are intended for variability instantiation rather than variability modelling, the Configuration types neither support changing the underlying IVML model nor its configuration.

3.4.4.1 IvmlElement

The `IvmlElement` is the most common configuration type. All configuration types discussed in this section are subclasses of `IvmlElement`. Further, this type represents the IVML identifiers used in VIL scripts or templates.

- **Boolean ==(IvmlElement i)**
Returns whether *operand* is the same `IvmlElement` as *i*.
- **String getName () / String name ()**
Returns the (unqualified) name of the *operand*.
- **String getQualifiedName () / String qualifiedName ()**
Returns the unqualified name of the *operand*.
- **String getType () / String type ()**
Returns the (unqualified) name of the type of the *operand*.
- **String getQualifiedType () / String qualifiedType ()**
Returns the unqualified name of the type of the IVML element.
- **Attribute getAttribute (String name) / Attribute attribute (String name)**
Returns the attribute of the *operand* with given name.
- **Attribute getAttribute (IvmlElement element) / Attribute attribute (IvmlElement element)**
Returns the attribute of the *operand* matching the given IVML identifier.
- **AnyType getValue () / AnyType value ()**
Returns the (untyped) configuration value of the *operand*.
- **String getStringValue () / String stringValue ()**
Returns the configuration value of the *operand* as a `String`.
- **Boolean getBooleanValue () / Boolean booleanValue ()**
Returns the configuration value of the *operand* as a `Boolean`.
- **Integer getIntegerValue () / Integer integerValue ()**
Returns the configuration value of the *operand* as an `Integer`.
- **Real getRealValue () / Real realValue ()**
Returns the configuration value of the *operand* as a `Real`.
- **EnumValue getEnumValue () / Enum enumValue ()**
Returns the configuration value of the *operand* as an `EnumValue`.

An `IvmlElement` can automatically be converted to a `String` containing the name of the `IvmlElement`.

3.4.4.2 EnumValue

This subtype of `IvmlElement` represents an IVML enumeration value. This subtype of `IvmlElement` does not specify any additional operations.

3.4.4.3 DecisionVariable

This subtype of `IvmlElement` represents a configured IVML decision variable.

- **Sequence<DecisionVariable> variables()**
Returns the frozen nested variables of *operand*. Except for the IVML types container and compound, this sequence will always be empty.
- **Sequence<Attribute> attributes()**
Returns the frozen attributes of *operand*. Except for the IVML types container and compound, this sequence will always be empty.

3.4.4.4 Attribute

This subtype of `IvmlElement` represents a configured IVML attribute. This subtype of `IvmlElement` does not specify any additional operations.

3.4.4.5 IvmlDeclaration

This subtype of `IvmlElement` represents the type underlying an IVML decision variable. Instances of this type do not provide any configuration values rather than providing access to the structure of the represented type, e.g., the nested variable declarations in an IVML compound.

3.4.4.6 Configuration

The `Configuration` type provides access to the *frozen* configuration values as well as to the type declarations for a certain IVML model. In particular, the configuration type allows to create projections of a configuration, e.g., to select a subset of decision variables and specify further operations on that subset such as passing it to rules or to (one or more) instantiators. Although being an `IvmlElement`, a configuration instance will not provide access to values.

- **Sequence<DecisionVariable> variables()**
Returns the configured and frozen decision variables of *operand*.
- **Sequence<Attribute> attributes()**
Returns the configured and frozen attributes of *operand*.
- **Boolean isEmpty()**
Returns whether configuration in *operand* is empty, i.e., does not contain decision variables. This may occur due to the projection capabilities of this type.
- **DecisionVariable getName(String name) / DecisionVariable byName(String name)**
Returns the specified decision variable (if it exists).
- **Configuration selectByName(String namePattern)**

Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression applied on (qualified and unqualified) decision variable names.

- **Configuration selectByType(String typePattern)**
Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression applied on (qualified and unqualified) type names.
- **Configuration selectByAttribute(String namePattern)**
Returns a configuration as a projection of *operand* containing those decision variables, which are attributed by the attribute specified in terms of a Java regular expression applied on attribute names.
- **Configuration selectByAttribute(String name, AnyType value)**
Returns a configuration as a projection of *operand* containing those decision variables which are attributed by the specified attribute (in terms of an IVML identifier) and value.

3.4.5 Built-in Artefact Types and Artefact-related Types

In this section, we will discuss the built-in artefact types. Please note that the (meta model) of the artefact model is extensible, so that further as well as derived types may be added if needed.

3.4.5.1 Path

A path represents a relative or absolute file or folder. Paths are always relative to the containing project, in more detail to the containing artefact model and normalized in Unix notation (using the slash as path separator). Further, paths may be patterns and contain wildcards according to the ANT conventions [9].

- **String getPath() / String path()**
Returns a string representation of *operand*.
- **Boolean isPattern()**
Returns whether the *operand* is a pattern, i.e., whether it contains wildcards.
- **JavaPath toJavaPath()**
Explicitly converts the *operand* into a Java package path.
- **String toOSPath()**
Turns the *operand* into a relative operating system specific path.
- **String toOSAbsolutePath()**
Turns the *operand* into an absolute operating system specific path.
- **deleteAll()**
Deletes all artefacts in the *operand* path.
- **mkdir()**
Creates a directory from the *operand* path. This operation will fail if applied to a pattern.
- **Boolean matches(Path p)**
Returns whether the (pattern in) *operand* matches the given path.
- **Set<FileArtifact> selectByType(Type t)**
Returns those artefacts matching *operand* and complying to the given artefact type *t*.

- **Set<FileArtifact> selectAll()**
Returns all artefacts matching *operand*.
- **String +(String s)**
Returns the string concatenation of *operand* and the given String *s*.
- **Boolean exists()**
Returns whether the artefact denoted by the path exists. The operation will always return false in case of a pattern path.
- **delete()**
Deletes the underlying artefact. This operation will cause no effects on pattern paths.
- **String getName() / String name()**
Returns the name of the file part of the path or, in case of a pattern path, the entire pattern path.
- **Path rename(String name)**
Renames the underlying artefact and returns the related new path.

Paths can be constructed from a `String` using the explicit constructor. Typically, scripts shall rely on the automatic conversions from `String` to `Path` or from `Path` to `FileSystemArtifact`. A `Path` can be converted automatically into a `FolderArtifact`.

3.4.5.2 JavaPath

A subtype of `Path` representing Java package paths (separated by “.”).

3.4.5.3 Project

The project type encapsulates the physical location of a product line including all artefacts, in particular in terms of an Eclipse project. There are no explicit constructors for this type, as instances will be provided by the VIL/EASy-Producer runtime environment.

- **String getName() / String name()**
Returns the name of the project in *operand*.
- **Set<FileArtifact> selectAllFiles()**
Returns all file artefacts in *operand*.
- **Set<FileArtifact> selectAllFolders()**
Returns all folder artefacts in *operand*.
- **Path getPath() / Path path()**
Returns the path the operand is located in.
- **Path localize(Project s, Path p)**
Localizes path *p* originally in project *s* to the project in *operand*.
- **Path localize(Project s, FileSystemArtifact a)**
Localizes path of *a* originally in project *s* to the project in *operand*. This operation does neither copy nor move *a*.
- **Set<FileArtifact> selectByType(Type t)**
Returns those file artifacts in the operand project which comply with the given type *t*.
- **Path getEasyFolder()**

Returns the path to the EASy producer configuration files in *operand*.

3.4.5.4 Text

Represents an artefact or a fragment artefact in terms of the underlying text and allows direct manipulations. The manipulations will be written back into the artefact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artefact.

- **Boolean isEmpty()**
Returns whether the text representation in *operand* is empty.
- **Boolean matches(String regEx)**
Returns whether the specified *regEx* matches the textual representation in *operand*.
- **Text substitute(String regEx, String r)**
Substitutes all occurrences of *regEx* in *operand* by *r* and returns the modified text.
- **Text replace(String s, String r)**
Substitutes all occurrences of *s* in *operand* by *r* and returns the modified text. This operation does not consider regular expression matches rather than direct text matches.
- **Text append(String s)**
Appends *s* to the end of *operand* and returns the modified text.
- **Text prepend(String s)**
Prepends *s* before the beginning of *operand* and returns the modified text.
- **Text append(Text t)**
Appends the entire contents of *t* to the end of *operand* and returns the modified text.
- **Text prepend(Text s)**
Prepends the entire contents of *t* before the beginning of *operand* and returns the modified text.

3.4.5.5 Binary

Represents an artefact or a fragment artefact in terms of the underlying binary form and allows direct manipulations. This type is subject to future extensions. The manipulations will be written back into the artefact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artefact.

- **Boolean isEmpty ()**
Returns whether the text representation in *operand* is empty.

3.4.5.6 Artifact

The most common artefact type. All specific artefact types are subtypes of *Artifact*.

- **delete()**

Delete the artifact in *operand* regardless of whether it is a file, a component, or a fragment within an artifact.

- **String getName () / String name()**
Returns the name of the artifact in *operand*. The specific meaning of the name depends on the actual artifact type.
- **Text getText() / Text text()**
Returns the textual representation of the artifact in *operand*. Whether the representation can be manipulated depends on whether the artifact itself may be modified.
- **Binary getBinary() / Binary binary()**
Returns the binary representation of the artifact in *operand*. Whether the representation can be manipulated depends on whether the artifact itself may be modified.
- **rename(String n)**
Renames the artifact in *operand*. The specific effect of this operation and whether it may be applied at all depends on the actual artifact type.

3.4.5.7 FileSystemArtifact

Represents the most common type of file system artefacts.

- **Path getPath() / Path path()**
Returns the path to *operand*.
- **move(FileSystemArtifact a)**
Moves the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.
- **copy(FileSystemArtifact a)**
Copies the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.

3.4.5.8 FolderArtifact

This type represents a folder in the file system and always belongs to a certain artifact model (and typically to a containing `Project`). `FolderArtifact` is a subtype of `FileSystemArtifact`.

- **Sequence<FileSystemArtifact> selectAll ()**
Returns all file system artifacts contained in *operand*.

`FolderArtifact` can automatically be converted into a `String` containing the path or into a `Path` denoting the location.

3.4.5.9 FileArtifact

This type represents a file in the file system and always belongs to a certain artefact model (and typically to a containing `Project`). `FileArtifact` is a subtype of `FileSystemArtifact`. Please note that the actual instance in a variable of type `FileArtifact` may belong to a subtype as the creation of artefact takes artefact specific rules into account.

A temporary `FileArtifact` can be constructed using the constructor without arguments. A string (containing a path) as well as a `Path` can be converted automatically into a `FileArtifact`.

A `FileArtifact` is created as the default fallback, i.e., if no more specific artefact matches the underlying real artefact. The artefact creation mechanism may be configured using an external Java properties file, which relates artefact names to file path patterns (overwriting or extending the built-in rules).

3.4.5.10 `VtlFileArtifact`

The `VtlFileArtifact` type is a subtype of `FileArtifact`. In particular, it helps the VTL template instantiator in dynamic dispatch between other types of file artefacts and actual VTL templates. It does not provide additional operations or conversions over the `FileArtifact`.

A `VtlFileArtifact` is created for all real file artefacts with file extension `vtl`.

3.4.5.11 `XmlFileArtifact`

The `XmlFileArtifact` is a built-in composite artefact, i.e., if it exists its content is analysed for known substructures, which are made available for querying and manipulation in terms of fragment artefacts.

- **`XmlElement getRoot() / XmlElement root()`**
Returns the root element of the XML artifact in *operand*.
- **`Set<XmlElement> selectAll()`**
Returns all XML elements contained in *operand*.

A `XmlFileArtifact` is created for all real file artefacts with file extension `xml`.

`XmlElement`

The `XmlElement` is a built-in fragment artefact, which belongs to the `XmlFileArtifact`. In particular, it inherits all operations from `Artifact` such as access to the representations of the contained text or CDATA.

- **`Set<XmlElement> selectAll()`**
Returns all XML elements contained in *operand*.
- **`Set<XmlAttributes> attributes()`**
Returns all XML attributes belonging to *operand*.
- **`XmlAttributes getAttribute(String n)`**
Returns the XML attribute in *operand* with the specified name *n*. Please note that this operation does not fail, if the specified attribute does not exist but rather the subsequent evaluation will stop.
- **`XmlAttributes setAttribute(String n, String v)`**
Defines or changes the XML attribute in *operand* with the specified name *n* to the given value *v*.
- **`Set<XmlAttribute> selectByName (String regEx)`**
Returns those XML attributes in *operand*, which comply with the given name pattern specified as Java regular expression.

XmlAttribute

The `XmlElement` is a built-in fragment artefact, which belongs to the `XmlFileArtifact` and to the fragment artefact `XmlElement`. In particular, it inherits all operations from `Artifact`.

- **String `getValue ()` / `String value()`**
Returns the value of the attribute in *operand*.
- **setValue (String `v`)**
Changes the value of the attribute in *operand* to *v*.

3.4.6 Built-in Instantiators

VIL provides also several built-in instantiators, in particular to modify or generate entire artefacts in one step. Basically, instantiators shall be defined using the VIL template language (this actually happens through an instantiator for VIL templates). However, very complex instantiation operations as well as existing (legacy) instantiator operations shall be available to the VIL build language, also as a better integrated alternative to just calling an operating system command. In this section, we will discuss the instantiators shipped with the VIL implementation as well as their specific operations. Please refer to the EASy-Producer developer documentation on how to realize custom instantiators.

3.4.6.1 VIL Template Processor

The VIL template processor is responsible for interpreting and executing VIL template scripts in close collaboration with the VIL build language. It may operate in two different modes depending on the actual arguments, namely executing VIL templates or replacing VIL expressions in a file artefact.

Basically, VIL templates receive three different parameters, the template, the variability configuration and the target artefact (fragment) to be produced. Thereby, the instantiator itself takes an argument of type `Artifact`, but the dynamic dispatch mechanism allows specifying subtypes in the template parameters or even to have multiple main subtemplates for different artefact types. In addition the VIL template processor may receive an arbitrary number of named arguments specific to the template to be executed.

This instantiator provides two instantiator operations:

- **Set<FileArtifact> `vilTemplateProcessor(VtlFileArtifact t, Configuration c, Artifact a, ...)`**
Parses and analyses the template in *t*, executes the template specification using the configuration *c* and replaces the content of *a*. Additional named arguments are passed to the VTL template *t* in order to customize the processing and must comply to the additional template parameters.
- **Set<FileArtifact> `vilTemplateProcessor(FileArtifact t, Configuration c, Artifact a)`**
Searches for VIL variables and expressions in *t* (variables given as `$variableName`, expressions as `${expression}`) and replaces them with their actual value as defined in the configuration *c*. The output replaces the content of *a*.

3.4.6.2 Blackbox Instantiators

In this section, we describe three built-in blackbox instantiators.

Velocity Instantiator

The Velocity⁷ instantiator [4] allows using one of the basic functionalities of EASy through VIL. It provides instantiator calls for individual templates and collections of templates.

- **Set<FileArtifact> velocity(FileArtifact t, Configuration c)**
Instantiates the template in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.
- **Set<FileArtifact> vilTemplateProcessor(Collection<FileArtifact> t, Configuration c)**
Instantiates the templates in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.

Java Compiler

The Java compiler blackbox instantiator allows to directly compile Java source code artefacts from the VIL build language. It provides the following instantiator call

- **Set<FileArtifact> javac(Path s, Path t, ...)**
Compiles the artefacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. Additional parameters of the Java compiler can directly be given as named attributes, such as a collection of Strings or Paths or Artefacts denoting the classpath, for example
sequenceOf(String) cp = {"\$source/lib/myLib.jar"};
Javac("\$source/**/*.*java", "\$target/bin", classpath=cp);

AspectJ Compiler

The AspectJ [1] compiler blackbox instantiator allows to directly compile Java and AspectJ source artefacts from the VIL build language. It provides the following instantiator call

- **Set<FileArtifact> aspectJ(Path s, Path t, ...)**
Compiles the artefacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. Additional parameters of the AspectJ compiler can directly be given as named attributes as described above for the Java compiler blackbox instantiator.

⁷ http://velocity.apache.org/engine/devel/user-guide.html#Velocity_Template_Language_VTL:_An_Introduction

4 VIL Grammars

In this section we depict the actual grammar for the VIL languages. The grammar is given in terms of a simplified xText⁸ grammar (close to ANTLR⁹ or EBNF). Simplified means, that we omitted technical details used in xText to properly generate the underlying EMF model as well as trailing “;” (replaced by empty lines in order to support readability). Please note that some statement-terminating semicolons are optional in order to support various user groups each having individual background in programming languages.

4.1 VIL Build Language Grammar

ImplementationUnit:

```
    Import*  
    LanguageUnit*;
```

LanguageUnit:

```
    Advice* 'vilScript' Identifier '(' ParameterList? ')'  
    (ScriptParentDecl)? '{'  
        VersionStmt?  
        LoadProperties*  
        ScriptContents  
    '}' ';'?
```

ScriptParentDecl:

```
    'extends' Identifier
```

LoadProperties:

```
    'load' 'properties' STRING ';'?
```

ScriptContents:

```
    (VariableDeclaration | RuleDeclaration)*
```

RuleDeclaration:

```
    (RuleModifier? Identifier '(' (ParameterList)? ')' '=')?  
    (LogicalExpression)? ':'  
    (LogicalExpression (',' LogicalExpression)*)?  
    RuleElementBlock ';'?
```

⁸ <http://www.eclipse.org/Xtext/>

⁹ <http://www.antlr.org>

RuleElementBlock:

'{' RuleElement* '}'

RuleElement:

VariableDeclaration
| MapStatement
| ExpressionStatement
| DeferDeclaration

RuleModifier:

'protected'

MapStatement:

'map' '(' Identifier (',' Identifier)* '=' Expression ')' '
RuleElementBlock ';'?

PrimaryExpression:

ExpressionOrQualifiedExecution
| UnqualifiedExecution
| SuperExecution
| SystemExecution
| Join
| ConstructorExecution

Join:

'join' '(' JoinVariable ',' JoinVariable ')' '
('with' '(' Expression ')')?

JoinVariable:

'exclude'? Identifier ':' Expression

SystemExecution:

'execute' Call SubCall*

4.2 VIL Template Language Grammar

```

Import*
Extension*
LanguageUnit:
    Advice*
    IndentationHint?
    'template' Identifier '(' ParameterList? ')'
    ('extends' Identifier)? '{'
        VersionStmt?
        VariableDeclaration*
        VilDef*
    '}'

IndentationHint:
    '@indent' '(' IndentationHintPart (',' IndentationHintPart)* ')'

IndentationHintPart:
    Identifier '=' NUMBER

VilDef:
    'def' Type? Identifier '(' ParameterList? ')' StmtBlock ';'

StmtBlock:
    '{' Stmt* '}'

Stmt:
    VariableDeclaration
    | Alternative
    | Switch
    | StmtBlock
    | Loop
    | ExpressionStatement
    | Content

Alternative:
    'if' '(' Expression ')' Stmt (=> 'else' Stmt)?

Content:
    (STRING) ('|' Expression ';')?

```


Switch:

```
'switch' '(' Expression ')' '{'
  (SwitchPart (',' SwitchPart)* (',' 'default' ':' Expression)?)
'{'
```

SwitchPart:

```
Expression ':' Expression
```

Loop:

```
'for' '(' Type Identifier ':' Expression
  (',' PrimaryExpression) ')' Stmt
```

Extension:

```
'extension' JavaQualifiedName ';' 
```

JavaQualifiedName:

```
Identifier ('.' Identifier)*
```

4.3 Common Expression Language Grammar

Actually, parts of this common language are overridden and redefined by the two VIL language grammars.

LanguageUnit:

```
Import*
Advice*
Identifier
VersionStmt?
```

VariableDeclaration:

```
'const'? Type Identifier ('=' Expression)? ';' 
```

Advice:

```
'@advice' '(' QualifiedName ')' VersionSpec?
```

VersionSpec:

```
'with' '(' VersionedId (',' VersionedId)* ')
```

VersionedId:

```
'version' VersionOperator VERSION
```

VersionOperator:

'==' | '>' | '<' | '>=' | '<='

ParameterList:

(Parameter (',' Parameter)*)

Parameter:

Type Identifier

VersionStmt:

'version' VERSION ';'

Import:

'import' Identifier VersionSpec? ';'

ExpressionStatement:

(Identifier '=')? Expression ';'

Expression:

LogicalExpression | ContainerInitializer

LogicalExpression:

EqualityExpression LogicalExpressionPart*

LogicalExpressionPart:

LogicalOperator EqualityExpression

LogicalOperator:

'and' | 'or' | 'xor'

EqualityExpression:

RelationalExpression EqualityExpressionPart?

EqualityExpressionPart:

EqualityOperator RelationalExpression

EqualityOperator:

'==' | '<>' | '!='

RelationalExpression:

AdditiveExpression RelationalExpressionPart?

RelationalExpressionPart:

RelationalOperator AdditiveExpression

RelationalOperator:

'>' | '<' | '>=' | '<='

AdditiveExpression:

MultiplicativeExpression AdditiveExpressionPart*

AdditiveExpressionPart:

AdditiveOperator MultiplicativeExpression

AdditiveOperator:

'+' | '-'

MultiplicativeExpression:

UnaryExpression MultiplicativeExpressionPart?

MultiplicativeExpressionPart:

MultiplicativeOperator UnaryExpression

MultiplicativeOperator:

'*' | '/'

UnaryExpression:

UnaryOperator? PostfixExpression

UnaryOperator:

'not' | '!' | '-'

PostfixExpression: // for extension

PrimaryExpression

PrimaryExpression:

ExpressionOrQualifiedExecution

| UnqualifiedExecution

| SuperExecution
| ConstructorExecution

ExpressionOrQualifiedExecution:

(Constant | '(' Expression ')') SubCall*

UnqualifiedExecution:

Call SubCall*

SuperExecution:

'super' '.' Call SubCall*

ConstructorExecution:

'new' Type '(' ArgumentList? ')' SubCall*

SubCall:

'.' Call | '[' Expression ']'

Call:

Identifier '(' param=ArgumentList? ')'

ArgumentList:

NamedArgument (',' NamedArgument)*

NamedArgument:

(Identifier '=')? Expression

QualifiedName:

Identifier (':' Identifier)* ('.' Identifier)?

Constant:

NumValue | STRING | QualifiedName | ('true' | 'false')

NumValue :

NUMBER

Identifier:

ID

Type:

```
Identifier
| ('setOf' TypeParameters)
| ('sequenceOf' TypeParameters)
| ('mapOf' TypeParameters)
```

TypeParameters:

```
(' Identifier (',' Identifier)* ')
```

ContainerInitializer:

```
{ (ContainerInitializerExpression
  (',' ContainerInitializerExpression)* )? }
```

ContainerInitializerExpression:

```
LogicalExpression | ContainerInitializer
```

terminal VERSION:

```
'v' ('0'..'9')+ ( '.' ('0'..'9') ) *
```

terminal ID:

```
('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9') *
```

terminal NUMBER:

```
'-'?
( ('0'..'9')+ ( '.' ('0'..'9') * EXPONENT? ) )?
| '.' ('0'..'9')+ EXPONENT?
| ('0'..'9')+ EXPONENT
```

terminal EXPONENT:

```
('e'|'E') ('+'|'-')? ('0'..'9')+
```

terminal STRING :

```
""
( '\\ ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'"'|'\\') | !('\\\\'|'"') ) *
"" | ""
( '\\ ('b'|'t'|'n'|'f'|'r'|'u'|'"'|'"'|'\\') | !('\\\\'|'"') ) *
""
```

terminal ML_COMMENT:

```
'/*' -> '*/'
```

terminal SL_COMMENT:

'/'/' !('\n'|'\r')* ('\r'? '\n')?

terminal WS:

(' '|'\t'|'\r'|'\n')+

terminal ANY_OTHER:

.

References

- [1] Project homepage AspectJ, 2011. Online available at: <http://www.eclipse.org/aspectj/>.
- [2] Eclipse Foundation. Xtend - Modernize Java, 2013. Online available at: <http://www.eclipse.org/xtend>.
- [3] INDENICA Consortium. Deliverable D2.1: Open Variability Modelling Approach for Service Ecosystems. Technical report, 2011.
- [4] INDENICA Consortium. Deliverable D2.4.1: Variability Engineering Tool (interim). Technical report, 2012.
- [5] INDENICA Consortium. Deliverable D2.2.2: Variability Implementation Techniques for Platforms and Services (final). Technical report, 2013.
- [6] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: <http://www.omg.org/docs/formal/06-05-01.pdf>.
- [7] SSE. Ivml language specification. http://projects.sse.uni-hildesheim.de/easy/docs/ivml_spec.pdf [validated: March 2013].
- [8] Richard M. Stallmann, Roland McGrath, and Paul D. Smith. GNU Make - A Program for Directing Recompilation - GNU make Version 3.82, 2010. Online available at: <http://www.gnu.org/software/make/manual/make.pdf>.
- [9] The Apache Software Foundation. Apache Ant 1.8.2 Manual, 2013. Online available at: <http://ant.apache.org/manual/index.html>.