Stiftung University of Hildesheim
Universitätsplatz 1
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

Preview version of 02.05.2024

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
1.1	19.02.2018	Standalone usage, new Eclipse classloading, EASy system properties and configuration files. Added hints to Maven Central after EASy release.
1.2	30.5.2024	Reasoner implementation description updated. EasyExecutor added. Migration to JDK 17 and Eclipse 2024-03. Revision of the example instantiator implementation.

Acknowledgements

We are grateful to Kevin Schaperjahn who pointed out several inconsistencies in this document, which were not cleaned up over time, in particular not after renaming EASy-Producer packages shortly after making EASy-Producer available on github.

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

EASy-Producer[1] 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. Further, 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 2.2 will give some technical recommendations, while Section 2.3. 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]. Since EASy-Producer version 1.3.10, in general, an Eclipse installation for JDK 17 with corresponding Xtext version at least 2.34 is fine for installing and running EASy-Producer. However, we cannot guarantee that any combination of Eclipse and Xtext will work with EASy-Producer. Thus, we propose Eclipse 2024-03 (3.41) and Xtext version 2.34

Until the next release, we recomment the EASy-Producer nightly update site http://projects.sse.uni-hildesheim.de/eclipse/update-sites/easy_nightly/ or the pre-packaged Eclipse versions on http://projects.sse.uni-hildesheim.de/eclipse/easy-nightly

2.2. Technical Recommendations

Since version 1.3.10, EASy-Producer is based on JDK 17 (17.0.2), maven (3.9.6), maven Tycho (4.0.7) and Eclipse (3.41).

2.3. 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 and 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**.

[2] Eclipse website: www.eclipse.org/
[3] Xtext website: http://www.eclipse.org/Xtext/

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 Services[4]. 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 3.3 will describe the implementation and integration of a new artefact type, while in Section 3.4, we will implement and integrate a new reasoner. Each section will provide detailed guidance from project creation and configuration to the final intragetion 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.

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. Using Additional Libraries

Eclipse changed its implementation of OSGI (equinox) over time, so that also the way of loading classes and providing access to required classes changed effectively. In particular, in Eclipse versions around 4.7 class loading became much more strict. So the legacy EASy way of packaging an instantiator with its libraries and keeping all library classes does not work anymore for all kinds of bundles, in particular if bundles are subject to dynamic class loading and reflection as it is the case for VIL. Before implementing an EASy-Producer extension, please think carefully about whether additional classes are needed at all, whether the classes are already used and provided by EASy (may be changing some runtime export directions in basic EASy bundles would help),whether they can be obtained from an (installed, required) Eclipse bundle or whether you have to provide them.

[4] For more information visit: http://eclipse.org/equinox/

If classes are already provided by EASy-Bundles, declare that bundle as required and, if you go for tests of your extension in an extra bundle (yes you should), **re-export** that dependency. It may be that the test bundle then must import the required packages.

If you need libraries that are not provided at all (or you want to use a specific version and keep it), you must create an **extra bundle just providing these classes**, i.e., call New|Project|Plug-in Development|Plug-in from Existing JAR Archives, specify those archives and create a plugin for those. Please review carefully the exported packages, i.e., whether really all of them are needed or whether they could cause conflicts with Easy/Eclipse, e.g., several libraries have a conflicting logging mechanism, rely on apache commons bundled with them, provide core parts of Eclipse, etc. Declare then that package as required in your bundle implementing the extension and, if needed, re-export the library packages.

Due to the Java module system introduced with Java 9-17, some **reflection accesses may be denied**. For the VIL core, e.g., we included Apache Xalan as local library so that we can manipulate the sequence of XML nodes in order to keep them as they were in the orignal artifact. On purpose, this library remains local and shall not be exported or even re-exported.

Due to the **Maven Tycho** based build process, each component now ships with an own Maven build process, mainly based on the central Maven dependencies of EASy-Producer. Where possible, please keep additional libraries local and also declare their versions in the respective build process. If dependencies are of global interest, they may be added to the central Maven dependencies of EASy-Producer. Keeping both, Maven and OSGi dependencies is the basis for the Eclipse target platform definition in the build process (possibly accompanied by an explicit target definition file). On the one side, this allows for a smooth automated build of EASy-Producer up until fully installed Eclipse RCP apps. On the other side, it allows for using EASy-Producer components standalone as Maven artifacts. For compliance with older Eclipse conventions, the EASy-Producer components still follow a `src/bin` structure rather than the usual Maven structure. This just requires some specific settings in the Maven POM, while all other Eclipse settings can be applied as discussed below. However, it's important to note that the OSGI bundle must have the same version number as the Maven POM (defined for EASy-Producer in the central Maven dependencies) – to reflect the Maven SNAPSHOT character, the OSGI bundle version must end with the `.qualifier` postfix. Further, the Maven `artifactId` must be the same as the OSGI bundle (symbolic) name.

Due to historical reasons and the execution in our continuous integration, we **currently do not provide a full Maven build process** consisting of a top-level maven project (central dependencies) and sub-ordinate aggregator POMs. For this purpose, all EASy-Producer components still contain a small ANT file triggering Maven and the automated deployment to our snapshot Maven repository.

3.2. 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

configuration[5]. 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 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 Dynamic 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.

3.2.1. 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 3.3). For further details about the language concepts, the available types, and their application, please considere the VIL language specification (cf. Section 2.3).

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 1.

[5] Details on the Velocity instantiator can be found in the EASy-Producer user guide (cf. Section 2.3).

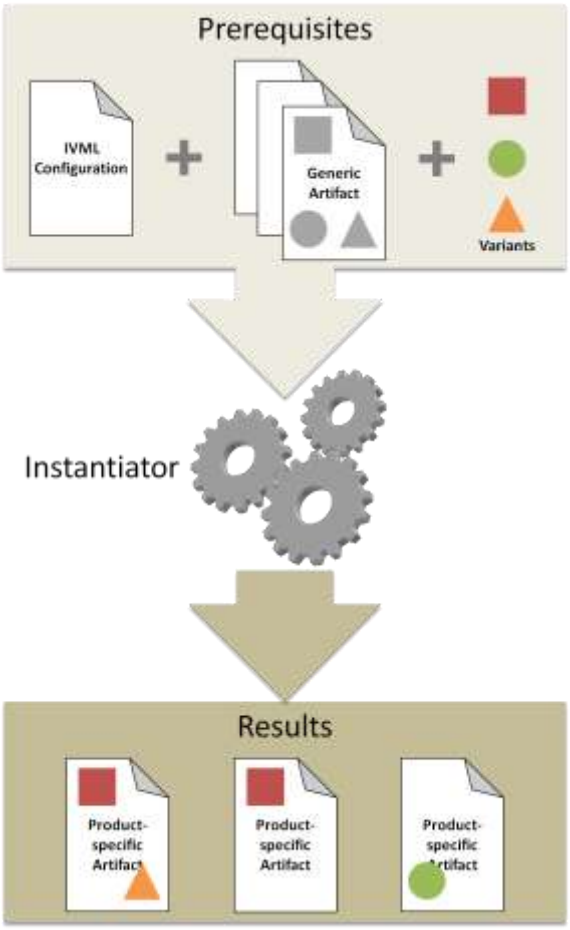


Figure 1: Black-box view of a generic instantiator (simplified)

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 1). 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 1) to which one or multiple variants (indicated as colored shapes in Figure 1) 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 2. 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 3.2.3). The way of processing this information depends on the implemented instantiator logic. Figure 2 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 contanier, i.e. in terms of searching for variation points in each file (this is described as VIT-statement in Figure 2; 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.

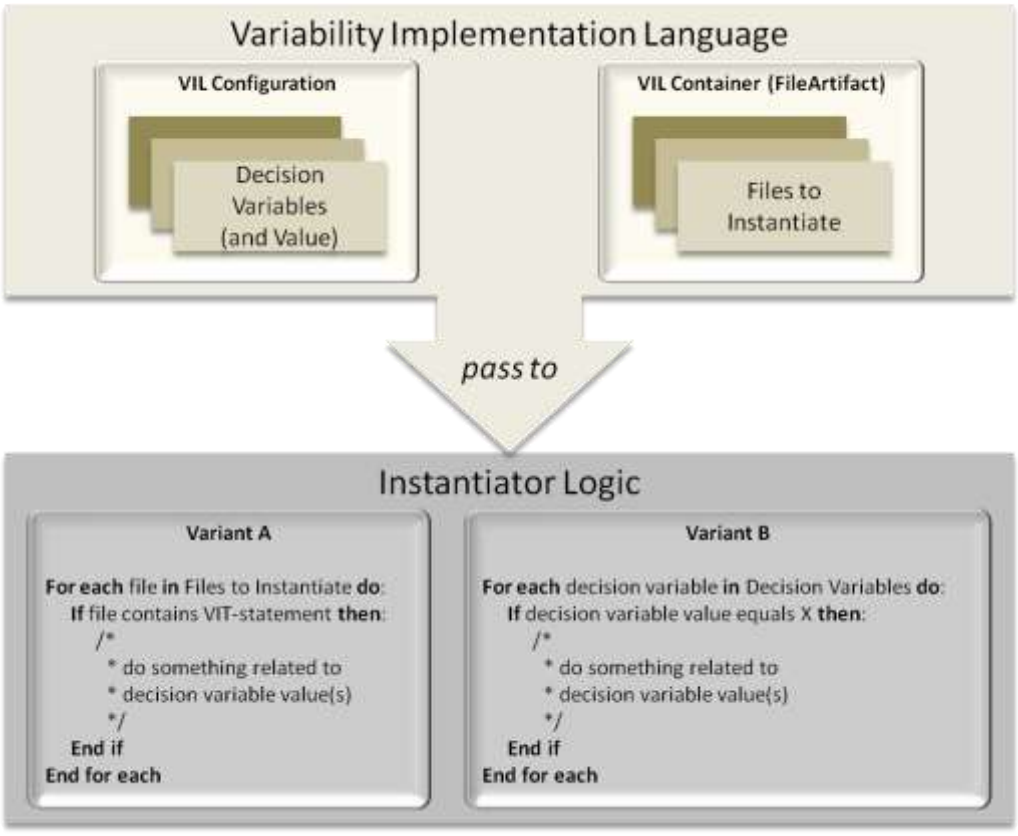


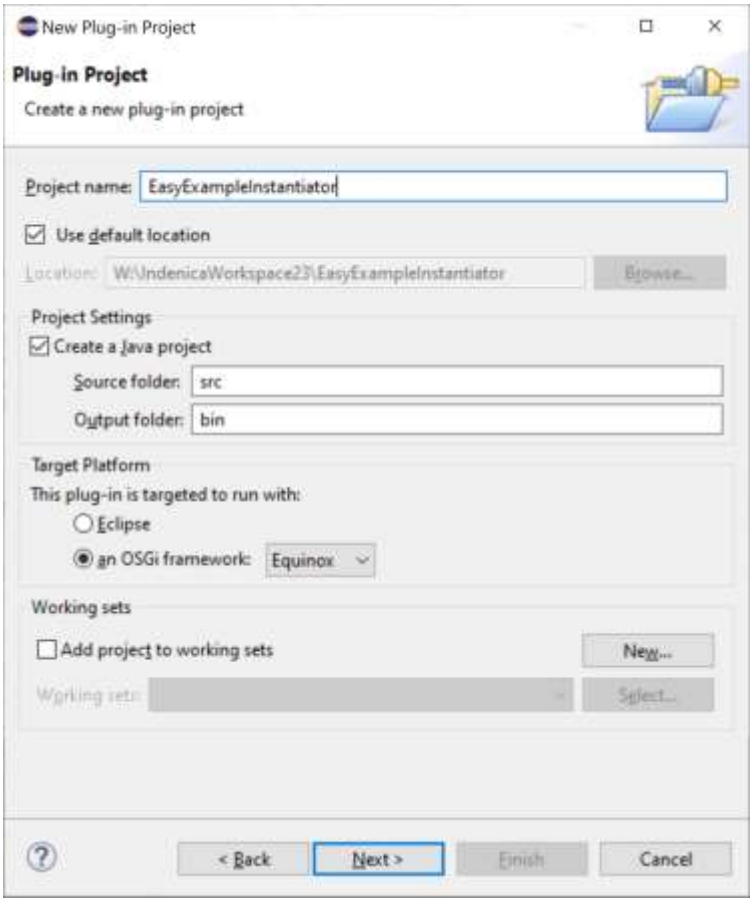
Figure 2: White-box view of a generic instantiator (simplified)

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.

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

Although we explain in this section the creation of a new instantiator step-by-step, we recommend obtaining a simple instantiator implementation like the velocity or the XVCL from the EASY-Producer github and adjusting this towards your needs.

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 3 shows how the first configuration page for the new plug-in project must look like.



Click on the *Next* button and define the properties of your plug-in. We will use the following values for the required properties[6]:

- *ID*: de.uni_hildesheim.sse.easy.instantiator.exampleInstantiator
- *Version*: 1.3.10.qualifier
- *Name*: EASyExampleInstantiator
- *Vendor*: 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:

- *net.ssehub.easy.instantiation.core*: This plug-in provides the core capabilities of the EASy-Producer instantiator concept. We will use parts of this plug-in in Section 3.2.3.
- *org.apache.felix.scr*: This plug-in may be needed for OSGI purposes and shall be set to optional resolution.

Further, click the *Add…* button for imported packages 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[7]. After adding the plugins/imports, please open their properties and remove the automatically added version number. Plugin versions will be determined by the target Platform of Maven Tycho. The *Dependencies* tab should now look like the one illustrated by Figure 4.

[6] The example uses version `1.3.10.qualifier`. As explained above, for Maven Tycho, the version must be the actual EASy version number followed by `.qualifier`.
[7] For more information visit: OSGi API – org.osgi.service.component

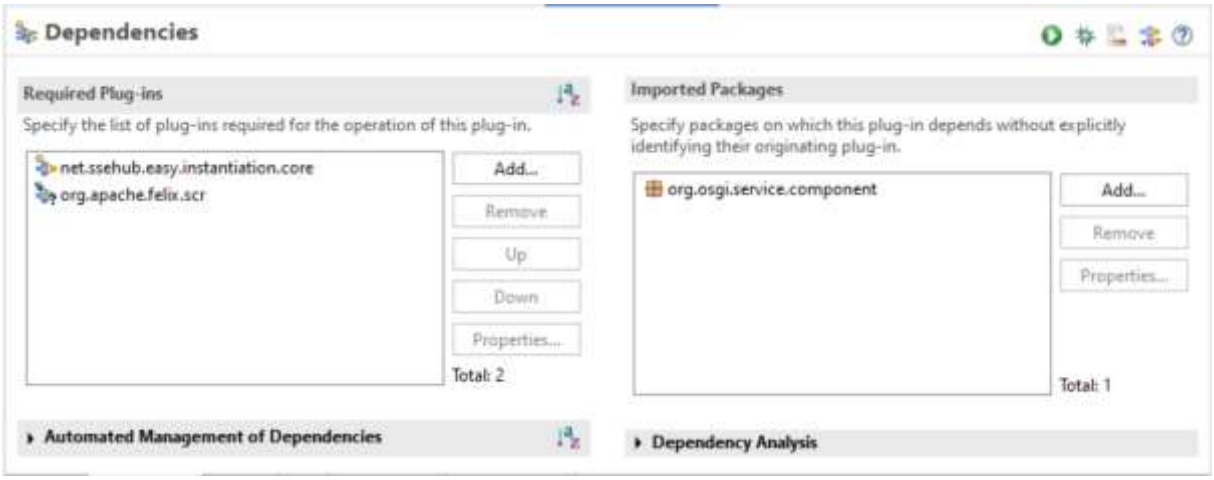


Figure 4: Definition of the required plug-ins for the new instantiator.

If needed, you may add further plugins, in particular library plugins such as `org.eclipse.text` or one of the libraries provided by EASy for compatibility such as `net.ssehub.easy.libs.commons.io`, `net.ssehub.easy.libs.commons.codec`, `net.ssehub.easy.libs.commons.lang`, `net.ssehub.easy.libs.slf4j.api`. Please note that it may be dangerous to assume that certain libraries that are included in your Eclipse will also be present in other Eclipse installations of the same or of future versions. Thus, it is safer to rely on provided libraries although the might be currently outdated (central updates may follow) or further provided libraries have to be added. In contrast to EASy-Producer, these library plugins are typically identified by their respective version rather than the actual EASy-Producer version.

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
```

as well as the following activation policy

```
Bundle-ActivationPolicy: lazy
```

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 5 shows how the manifest file must look like.

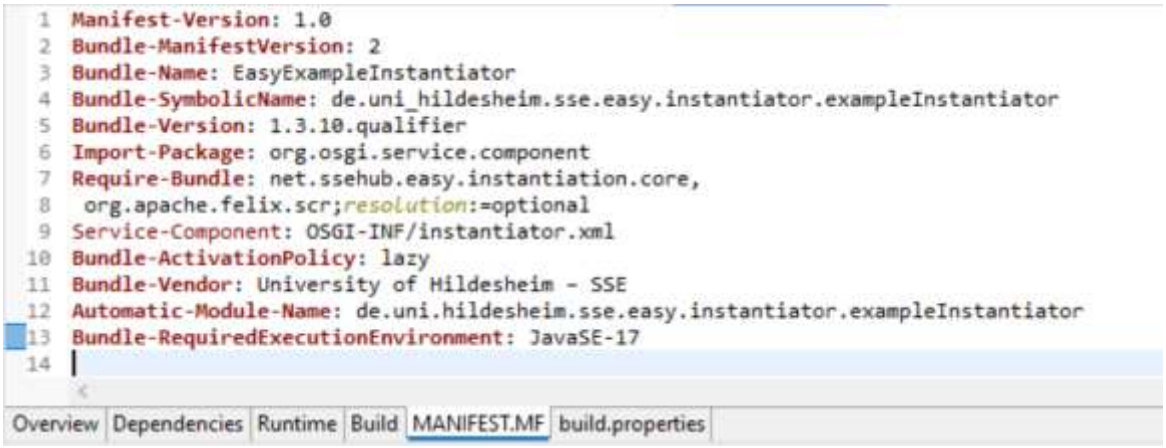


Figure 5: Declaration of the service-component for the new instantiator

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*[8], 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 5: *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"?>

<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 6 shows the final XML file. Please note that we used the names and package-structure of our example in Figure 6. 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 3.2.3.

[8] 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.

Figure 6: 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 first[9]. 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 7 and Figure 8 show the result in the context of our example. Figure 7 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 8 illustrates the build configuration in which the library (highlighted) is selected as part of the *Binary Build*.

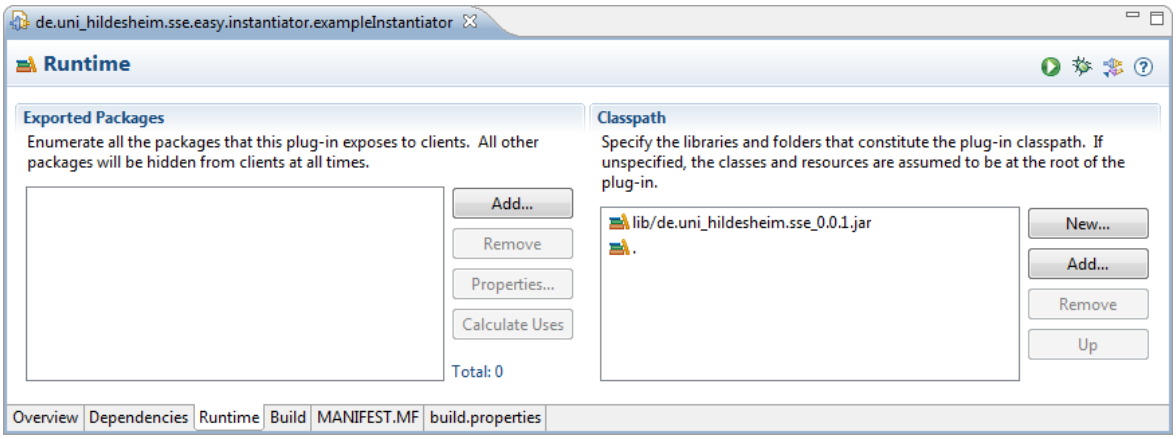


Figure 7: Classpath specification of external, required libraries

[9] If you do not know how to build a plug-in, please consider Section 3.2.3.

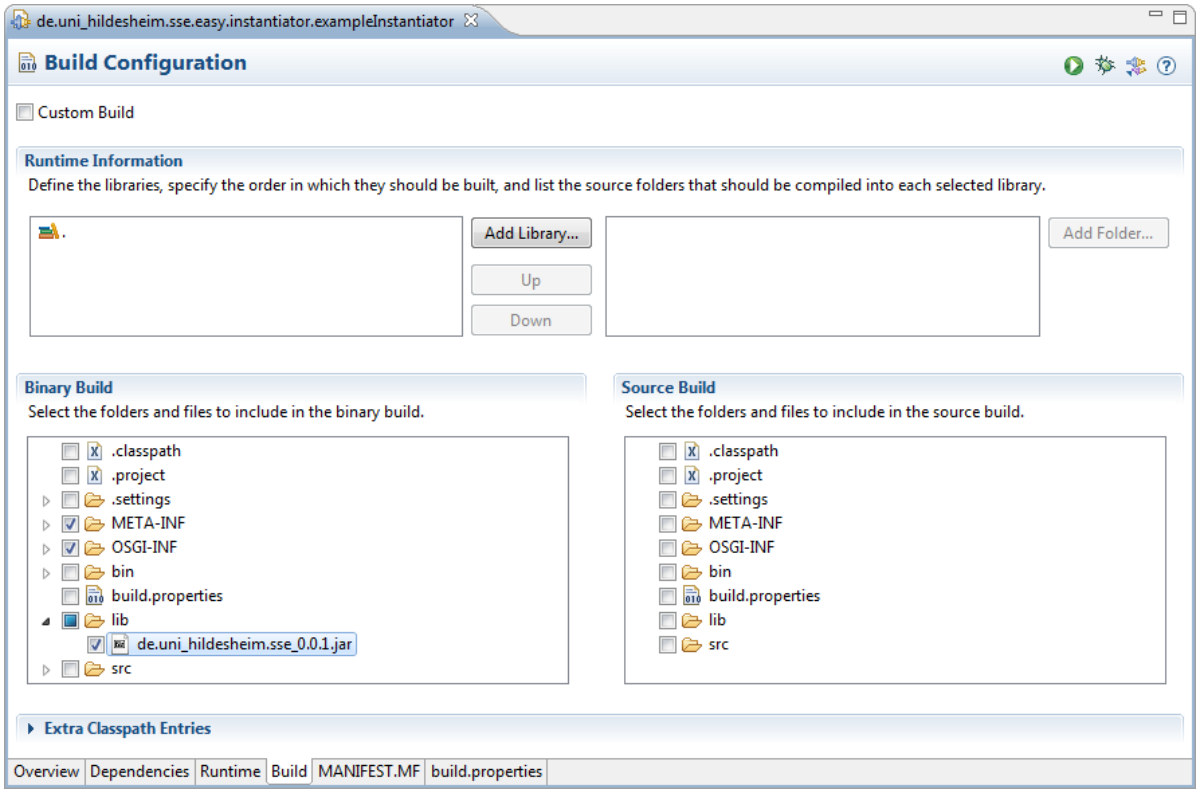


Figure 8: Binary Build selection of external, required libraries

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.

3.2.3. 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 3.2.2 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 3.2.2). 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 3.2.4, while details about VIL types in gereral, the annotation, and, in particular, the *IVilType* interface can be obtained in the VIL language specification (cf. Section 2.3).

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 org.osgi.service.component.ComponentContext;

import net.ssehub.easy.instantiation.core.model.artifactModel.FileArtifact;
import net.ssehub.easy.instantiation.core.model.common.VilException;
import net.ssehub.easy.instantiation.core.model.vilTypes.Collection;
import net.ssehub.easy.instantiation.core.model.vilTypes.IVilType;
import net.ssehub.easy.instantiation.core.model.vilTypes.Instantiator;
import net.ssehub.easy.instantiation.core.model.vilTypes.Set;
import net.ssehub.easy.instantiation.core.model.vilTypes.TypeRegistry;
import net.ssehub.easy.instantiation.core.model.vilTypes.configuration.Configuration;

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

  protected void activate(ComponentContext context) {
        TypeRegistry.DEFAULT.register(ExampleEngine.class);
  }

  protected void deactivate(ComponentContext context) {
  }

  public static Set<FileArtifact> exampleEngine(Collection<FileArtifact>
        templates, Configuration config) throws VilException {
        // 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 register the new type in the VIL type registry. This will include the type in the **Artifact Model** of VIL (see VIL Language Specification for details on the VIL 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 *exampleEngine* (with literally the name as given to the instantiator in the *Instantiator* class annotation) will be used in the VIL language to call this instantiator, including the defined parameters. In our example, this method requires the follwing 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 VIL Artifact Model. This set of artefacts will be processed by the instantor 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 example 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 destination[10], and click on the *Finish* button.

Finally, the plug-in and, thus, the instantiator is implemented, build, and ready for (local) 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.

For building the instantiator as part of the EASy-Producer continuous build process, the instantiator also needs a Maven Tycho build specification and (still) a simple ANT build file. The Maven build specification for the example instantiator would look as follows (assuming that net.ssehib.easy:dependencies was installed locally through Maven, either from the SSE maven repository or through the source code from github)

```
<?xml version="1.0" encoding="UTF-8"?><project
xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <artifactId>net.ssehub.easy.instantiation.easyExampleInstantiator</artifactId>
  <packaging>eclipse-plugin</packaging>

  <name>EASy-Producer Example instantiator</name>
  <description>An Example instantiator of EASy-Producer.</description>
  <url>https://sse.uni-hildesheim.de/forschung/projekte/easy-producer/</url>

  <parent>
     <groupId>net.ssehub.easy</groupId>
     <artifactId>dependencies</artifactId>
     <version>1.3.10-SNAPSHOT</version>
     <relativePath/>
  </parent>

  <build>
    <sourceDirectory>src</sourceDirectory>
    <directory>bin</directory>
  </build>
```

[10] 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 3.2.4.

```
  <dependencies>
     <dependency>
        <groupId>net.ssehub.easy</groupId>
        <artifactId>net.ssehub.easy.instantiation.core</artifactId>
        <version>${project.version}</version>
     </dependency>
     <!- add further, specific dependencies including those added locally to
         the instantiator -->
  </dependencies>
 </project>
```

and the corresponding ANT file (by convention named `build-jk.xml`) only for execution by the continous server looks like

```
<project name="net.ssehub.easy.instantiation.EasyExampleInstantiator"
  default="all" basedir=".">
    <property file="${user.home}/migration.properties"/>
    <property file="${user.home}/global-build.properties"/>
    <include file="${user.home}/macros.xml"/>

    <target name="all">
        <maven pomFile="pom.xml" goal="deploy"/>
    </target>
</project>
```

3.2.4. 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*. 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 3.2.3 and passing the required parameters. Figure 9 shows the call of our example instantiator.

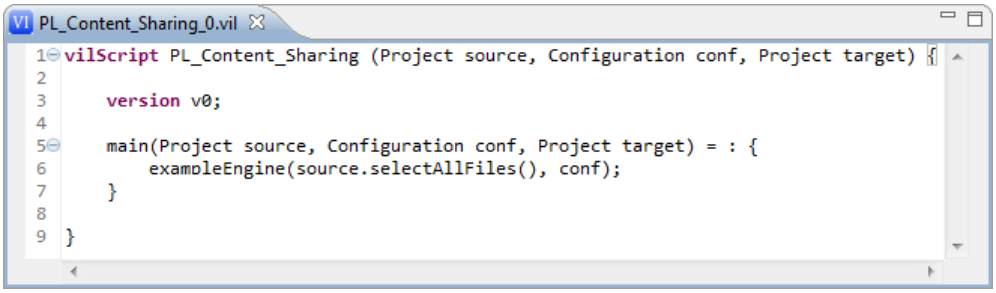


Figure 9: Using the new instantiator in a VIL build script.

3.3. 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.

3.3.1. 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 2.3).

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 illsutrate the usage of this interface in the implementation of our example in Section .3.3.3
- 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[11] **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 basic operation meta information such as renaming the operation (for operator implementations like for + or for names that would conflict with the Java method naming conventions).
- ***ReturnGenerics*** for making the type parameters of a generic return type explicit, which are otherwise removed/filtered out by the Java compiler.
- ***NonOclCompliant*** to mark operations for which the VIL/VTL editor shall issue a non-OCL compliant warning.
- ***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 [undbound] 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.

[11] Before since EASy-Producer 1.3.10 and JDK 17, OperationMeta was a complex annotation handling e.g., return generics and non-OCL warnings. With JDK 17 we were forced to split these aspects into individual annotations.

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 a **VilException**.

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.

3.3.2. 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 3.2.2. Below, we will decribe 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>
```

3.3.3. 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 3.3.1. We will extend the **FileArtifact** in this example. Thus, the content[12] of the new class file looks like this:

```
package easyexampleartifact;

import java.io.File;

import org.osgi.service.component.ComponentContext;

import net.ssehub.easy.instantiation.core.model.artifactModel.ArtifactCreator;
import net.ssehub.easy.instantiation.core.model.artifactModel.ArtifactModel;
import net.ssehub.easy.instantiation.core.model.artifactModel.FileArtifact;
```

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

```
import net.ssehub.easy.instantiation.core.model.artifactModel.FragmentArtifact;
import net.ssehub.easy.instantiation.core.model.artifactModel.IArtifactVisitor;
import net.ssehub.easy.instantiation.core.model.artifactModel.representation.Binary;
import net.ssehub.easy.instantiation.core.model.artifactModel.representation.Text;
import net.ssehub.easy.instantiation.core.model.vilTypes.ArtifactException;
import net.ssehub.easy.instantiation.core.model.vilTypes.IVilType;
import net.ssehub.easy.instantiation.core.model.vilTypes.Set;
import net.ssehub.easy.instantiation.core.model.vilTypes.TypeRegistry;
import net.ssehub.easy.instantiation.core.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) {
		TypeRegistry.register (ExampleArtifact.class);
	}

	protected void deactivate(ComponentContext context) {

	}

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

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

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

	@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 VilException {
		// Here goes the implementation
	}
```

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

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

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

	@Override
	public void store() throws VilException {
		// 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 3.3.1. 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 example, 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 net.ssehub.easy.instantiation.core.model.artifactModel.ArtifactModel;
import net.ssehub.easy.instantiation.core.model.artifactModel.DefaultFileArtifactCreator;
import net.ssehub.easy.instantiation.core.model.artifactModel.FileArtifact;
import net.ssehub.easy.instantiation.core.model.artifactModel.IArtifact;
import net.ssehub.easy.instantiation.core.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 VilException {
              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 fullfilled 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 example, 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 destination[13], and click on the *Finish* button.

Finally, the plug-in and, thus, the new artefact is implemented, build, and ready for local use. The integration is the same as in the instantiator case described in Section 3.2.4 (copy the final plug-in into the dropins-folder). The next time Eclipse will be started the new artefct type can be used in the VIL build language and in the VIL template language. As discussed in Section 3.2.3, a Maven and an ANT build file are required for continuous integration.

3.4. 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.

[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 3.2.4.

3.4.1. 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 3.2.2). 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*: 1.3.10.qualifier
- *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.osgi.service.component* (described in Section 3.2.2)
- *org.eclipse.core.runtime* (described in Section 3.2.2)
- *net.ssehub.easy.reasoning.core*: This plug-in provides the core capabilities of the EASy-Producer reasoning concept. We will use parts of this plug-in in Section 3.4.2.
- *net.ssehub.easy.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 3.4.2.

In the imports, 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. As explained for instantiators, please remove all version informations of all dependencies. No higher layers of EASy-Producer shall be used as they may introduce cyclic dependencies.

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
```

as well as the following activation policy

```
Bundle-ActivationPolicy: lazy
```

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*[14], 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 decribed in Section 3.2.2.

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.

3.4.2. 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.

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:

[14] 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.

```
package easyexamplereasoner;

import net.ssehub.easy.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);
		addCapability(IvmlReasonerCapabilities.ATTRIBUTES);
		addCapability(GeneralReasonerCapabilities.EVAL);
		// and all other provided capabilities
	}

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

	@Override
	public boolean providesAffectedVariables() {
		return false;
	}

}
```

Reasoner descriptors shall declare the capabililities of the described reasoner. Here, the developer shall be as honest as possible, not declaring any capabilitity that is not supported. The provided capabilities set helps EASy-Producer identifying an appropriate initial reasoner, whether there is a reasoner that can initialize a configuration or whether the user interface shall warn the user when certain operations / modeling concepts are utilized that are not supported by the actual reasoner selected by the user. In the example above, the reasoner just supports IVML attributes/annotations and eval-blocks but, e.g., no reasoning timeout, no configuration initialization, no IVML null values etc.

- *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 net.ssehub.easy.basics.progress.ProgressObserver;
import net.ssehub.easy.reasoning.core.frontend.IReasonerInstance;
import net.ssehub.easy.varModel.confModel.Configuration;
import net.ssehub.easy.varModel.model.AbstractVariable;
import net.ssehub.easy.varModel.model.Constraint;
import net.ssehub.easy.varModel.model.Project;
```

```
import net.ssehub.easy.varModel.model.datatypes.IDatatype;
import net.ssehub.easy.reasoning.core.frontend.ReasonerFrontend;
import net.ssehub.easy.reasoning.core.reasoner.EvaluationResult;
import net.ssehub.easy.reasoning.core.reasoner.IReasoner;
import net.ssehub.easy.reasoning.core.reasoner.IReasonerMessage;
import net.ssehub.easy.reasoning.core.reasoner.ReasonerConfiguration;
import net.ssehub.easy.reasoning.core.reasoner.ReasonerDescriptor;
import net.ssehub.easy.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 ReasoningResult check(
			Configuration cfg,
			ReasonerConfiguration reasonerConfiguration,
			ProgressObserver observer) {
		// Here goes the implementation
	}

	@Override
	public ReasoningResult propagate(
			Configuration cfg,
			ReasonerConfiguration reasonerConfiguration,
			ProgressObserver observer) {
		// Here goes the implementation
	}

	@Override
	public ReasoningResult initialize(
			Configuration cfg,
			ReasonerConfiguration reasonerConfiguration,
```

```
                ProgressObserver observer) {
        // Here goes the implementation
    }

    @Override
    public EvaluationResult evaluate(
                Configuration cfg,
                List<Constraint> constraints,
                ReasonerConfiguration reasonerConfiguration,
                ProgressObserver observer) {
        // Here goes the implementation
    }

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

    @Override
    public IReasonerInstance createInstance(
                Configuration cfg,
                ReasonerConfiguration reasonerConfiguration,
                ProgressObserver observer) {
        // Here goes the implementation
    }

    @Override
    public ValueCreationResult createValue(
                Configuration cfg,
                AbstractVariable var,
                IDatatype type,
                ReasonerConfiguration reasonerConfiguration,
                ProgressObserver observer) {
        // Here goes the implementation
    }

    @Override
    public void setInterceptor(IReasonerInterceptor interceptor) {
        // 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 invokes by EASy-Producer if a given variability model should be checked for satisfiability. However, the actual implementation of this method depends on the reasoner.
- ***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.
- ***initialize:*** This method initializes an initial Configuration, i.e., assigns values and processes constraints. This functionality is optional and shall be indicated in the reasoner capabilities in the reasoner descriptor. If indicated and the reasoner is active, EASy-Producer may automatically select this reasoner for constraint initialization if the capabilities outreach the capabilities of other available reasoners.
- ***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.
- ***notify:*** This method is called when a reasoner message is issued.
- ***createInstance:*** Allows for binding a reasoner instance to a configuration in order to enable incremental reasoning. If implemented, a reasoner may reuse internal structures and knowledge about a configuration and the underlying process to speed up repeaded reasoning requests. This capability is optional and handled by the `ReasonerAdapter` class, which selects an instance-based reasoner for a configuration if available or, as a fallback, obtains a new instance.
- ***createValue:*** This method creates IVML values for a given variable/type on the fly, in particular for runtime reasoning. This is required, as IVML values and their freezing may strongly depend on previously evaluated constraints.
- ***setInterceptor:*** Optional support for intercepting and observing the reasoning process, e.g., for debugging. A reasoner may take an interceptor and pass the control to the interceptor after evaluating a constraint.

Recently, we added capabilities to obtain a reasoner instance for a given project and reasoner configuration. Reasoner instances shall facilitate the re-use of internal structures in incremental reasoning and speed up reasoning operations. A reasoner instance may be interrupted by the user.

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 destination[15], 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.

[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 reasoner in Section 3.4.3.

3.4.3. Configuration initialization

A reasoned may support initializing a fresh configuration. EASy-Producer selects a reasoned with the capability of configuration initialization for this purpose with a precedence for the user-selected reasoned (see below). In case that there is no reasoner with configuration initialization capability available, EASy-Producer falls back to a simplified default configuration initialization functionality (focusing on non-dependent default and assignment constraints as well as annotation assignment blocks). If for some reasons it is required to force using the simplified initialization functionality, use the system property

```
-Deasy.configuration.useAssignmentResolver=true
```

3.4.4. 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 availabe as shown in Figure 10.

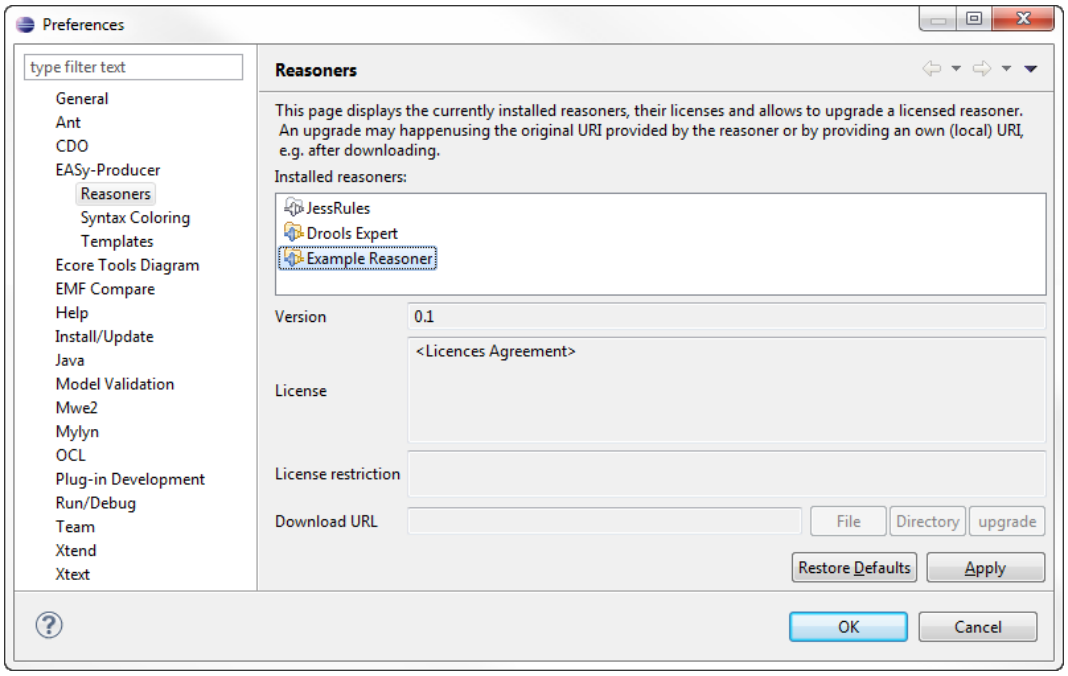


Figure 10: Reasoner preferences page

3.5. Running EASy/Bundles in Eclipse

Running a plugin within EASy either as jUnit Plugin test or as Eclipse plugin is not always easy, in particular if the underlying bundles are not explicitly defined by versions. If your Eclipse installation contains multiple versions of several core bundles, strange errors can occur, e.g., the

Eclipse SimpleConfigurator throws an illegal state exception that core bundles were updated and you should restart while restarting does not help resolving the problem.

Siince EASy-Producer 1.3.1 we provide an Eclipse Launch Configuration in the github repository that works for us. This Launch Configuration is accompanied with a ZIP file of plugins that are still missing in Eclipse. Please unpack these plugins into your Eclipse dropin folder. In case that you want to create your own launch configuration, you may follow the steps below:

- Try to run EASy-Producer.UI as Eclipse application. If this works, be happy. If not, there are multiple potential causes and even the resolution messages or exceptions may not always help. If running EASy-Producer directly fails, go on with the next steps.
- Check your Eclipse target platform (Window|Preferences|Plug-in Development|TargetPlatform) whether it contains errors or the most recent bundles. When updating Eclipse, the target platform is not updated and so outdated, unwanted or inconsistent bundles may be in your target platform. In the extreme case, try it with a new one (Add…, select Default).
- Go to your EASy-Producer.UI run configuration,
    - In the Configuration tab select "Clear the configuration area before launching"
    - In the Plugins tab select "plug-ins selected below". Deselect all plugins and select only net.ssehub.easy.producer.ui, then "Add required plugins".
    - Check whether org.eclipse.ui.ide.platform, org.osgi.service.component and org.eclipse.equinox.event are included, as they usually are included through optional dependencies or not at all. Also add com.google.guava even in multiple versions as xText may require a different version than Eclipse.
    - Add the top-level plugins/features for the bundles that you need in your runtime Eclipse, e.g., subversion, git or Maven.
    - Execute "Validate Plug-ins" and incrementally add plug-ins as needed. If you are brave, you can also request Eclipse to add required plugins
    - Try to add still missing plugins in your Eclipse dropins folder.

You can apply a simlar procedure to run the test cases as jUnit plugin test cases, starting with the respective test case. If you receive a strange error from the resources bundle indicating that it tries to read a non-XML language with an XML-Parser, the EASy-Resource initializer may believe that you are running with full Eclipse rather than as a test in a standalone mode. In this case, you must force EASy to go for standalone mode, just specifying `-Deasy.notInEclipse=true` (the value is irrelvant, only the property must be specified).

4. Using EASy-Producer Standalone

EASy-Producer is primarily implemented as an Eclipse tool, i.e., as also described above, its components are given in terms of Eclipse Plugins. At a glance, this prevents re-using EASy-Producer outside Eclipse. More precisely, architectural conventions preventing that basis components depend too much on Eclipse, additional code that allows Eclipse components such as JDT or xText to run outside Eclipse, and an emulation of the very core parts of Eclipse/OSGi-Interfaces and, in particular, the component loading mechanisms set up the environment that

core parts of EASy-Producer such as IVML, VIL, or the reasoning can be packaged and executed as a library outside Eclipse.

In this section, we first detail in Section 4.1 the core components of EASy-Producer, their application for re-use within an Eclipse-Plugin in Section 4.2, the additional mechanisms for re-using EASy-Producer outside Eclipse, in Section 4.3 using the pre-built libraries and through a self-defined composition using Maven.

4.1. The EASy-Producer Layers

As stated above, core components of EASy-Producer are implemented without or without too many dependencies to Eclipse. Obeying their internal dependencies, this allows creating further Eclipse plugins on top of the EASy-Producer core components. We discuss now the organization of these components into layers and the core components[16].

4.1.1. Foundations and object models layer

This layer contains the basic classes of EASy-Producer as well as the implementation of the "executable" object models for IVML (in terms of reasoning), VIL and VTL (in terms of execution for artifact manipulation or generation). While VIL can be seen as optional, the IVML model implementation is mandatory and a prerequisite for VIL/VTL. Please note that these packages only contain the models, not the parsers for the textual IVML/VIL langugages.

- `net.ssehub.easy.basics`: Defines common basic classes as well as the delegating EASy Logging Mechanism, serving for both, Eclipse-based and standalone logging. For working with IVML/VIL/VTL in a standalone manner, the most important (abstract) class is net.ssehub.easy.basics.modelManagement.ModelManagement. For a kind of model, this class knows how to load the model, e.g., through a parser, which models do exist and where, which models take precedence over others during import and which import resolution mechanism to use for a certain language. The language-dependent parts are implemented in the following components/bundles.
- `net.ssehub.easy.varModel`: Implements all classes relevant to represent and to persist an IVML model in textual manner. This includes in particular the class `net.ssehub.easy.varModel.model.Project` representing an IVML project, the IVML expression evaluation mechanism in `net.ssehub.easy.varModel.cstEvaluation.EvaluationVisitor` and the Configuration in `net.ssehub.easy.varModel.confModel.Configuration` assigning actually configured values to variable declarations in IVML projects. Using an IVML model always implies obtaining a `Project` (usually from the related model management class `net.ssehub.easy.varModel.VarModel`) and creating a `Configuration` for that project filling the configuration somehow with values. Nevertheless, models can be created on the fly without any parser and persisted if needed.

[16] Some components follow the new package naming scheme indicating the ssehub/github origin, some like the DSL languages still follow the old package naming as renaming xText languages is not just a simple refactoring operation.

- net.ssehub.easy.reasoning.core: Contains the basic interfaces for reasoning over IVML models. This includes the ReasonerFrontend as well as the interfaces for descriptors and reasoning as introduced in Section 3.4. However, this package does not contain any specific reasoner rather than mechansisms to register reasoners against and to retrieve reasoners.
- net.ssehub.easy.instantiation.core: Implements all classes representing the executable models of VIL and VTL. The main classes here are net.ssehub.easy.instantiation.core.model.buildlangModel.Script for VIL scripts (BuildModel in the same package is the VIL model management) and net.ssehub.easy.instantiation.core.model.templateModel.Template for VTL scripts (BuildTemplate in the same package is the VTL model management). Execution of a VIL script, and indirect all referenced VTL scripts) happens through interpretive model visitors. However, it is advisable not to utilize these visitors directly, rather than the central VIL model executor net.ssehub.easy.instantiation.core.model.execution.Executor.

4.1.2. DSL languages, models and parsers

Typically, creating models through the respective classes and constructors is only needed for very low-level tests or for specific user support operations. Mostly, IVML and VIL/VTL are utilized through their textual DSL languages. These components are implemented using xText and, thus, imply xText dependencies. Following xText conventions, each languages is implemented in three bundles 1) the parser, which can be used standalone 2) the tests (in our case coarse-grained language tests utilizing the parser and the translation into the object models) 3) the user interfaces, which is not considerd to be a logical part of this layer rather than the UI layer. The core components here are

- net.ssehub.easy.dslCore: Common mechanisms for all EASy-Producer languages, including functions setting up the right (standalone or Eclipse) environment for xText languages as well as some basic language testing support.
- de.uni_hildesheim.sse.ivml: The IVML grammar, the IVML parser and the language translation to instances in net.ssehub.easy.varModel.
- de.uni_hildesheim.sse.vil.expressions: The common VIL/VTL expression language including grammer, parser fragments, and common language translation to the respective classes in net.ssehub.easy.instantiation.core.
- de.uni_hildesheim.sse.vil.buildlang: The VIL grammar, the VIL parser and the language language translation to the respective classes in net.ssehub.easy.instantiation.core. This component depends on de.uni_hildesheim.sse.vil.expressions.
- de.uni_hildesheim.sse.vil.templatelang: The VTL grammar, the VTL parser and the language language translation to the respective classes in net.ssehub.easy.instantiation.core. This component depends on de.uni_hildesheim.sse.vil.expressions.

The parsers are registered through OSGi descriptive services with the respective model management classes, i.e., asking a model management implementation for the available models and loading a model involves functionality provided by the xText parsers.

4.1.3. IVML reasoning

Reasoners for IVML reasoning rely on `net.ssehub.easy.varModel` implement the interfaces in `net.ssehub.easy.reasoning.core` and register themselves through OSGi descriptive services against the `ReasonerFrontend`. Reasoners are called exclusively through the `ReasonerFrontend`. As various different reasoners providing different capabilities may exist (SAT-based, rule-based or direct implementations have been developed over the time), IVML reasoning is considered as an extension to the models layer rather than a part of the very core. Recently, most of the existing reasoners have been discontinued due to performance or IVML completeness issues. The currently only remaining reasoner is

- `net.ssehub.easy.reasoning.sseReasoner`: A fast forward-chaining reasoning mechanism with value propagation based on the IVML expression evaluation mechanism in `net.ssehub.easy.varModel`.

4.1.4. VIL instantiators

The core implementation of VIL does not contain language specific instantiators or types (except for XML, which we consider as rather common and generic). Instantiators depend on `net.ssehub.easy.instantiation.core`. EASy-Producers ships with several instantiators, most providing additional functionality for Java-based product line implementations. These instantiators are implemented based on the extension mechanisms discussed in Sections 3.1 and 3.3, all aim at providing a certain independent instantiation functionality, which is hooked through OSGi descriptive services into the VIL/VTL core. However, as the VIL/VTL core, all instantiators can contribute not only implementation but also VIL/VTL code to the default VIL/VTL library. Recently, the following VIL instantiators do exist:

- `net.ssehub.easy.instantiation.java`: Integration of the Java compiler as well as Java code manipulation mechanisms.
- `net.ssehub.easy.instantiation.aspectj`: Integration of the AspectJ compiler.
- `net.ssehub.easy.instantiation.ant`: Integration of the ANT build mechanism for Java.
- `net.ssehub.easy.instantiation.maven`: Integration of the Maven build mechanism for Java.
- `net.ssehub.easy.instantiation.velocity`: Integration of the Apache conditional preprocessor.
- `net.ssehub.easy.instantiation.xvcl`: Integration of the XVCL product line artifact approach.
- `net.ssehub.easy.instantiation.docker`: Integration of the main Docker commands.
- `net.ssehub.easy.instantiation.lxc`: Integration of the main LXC commands.

4.1.5. Model persistence

Sometimes, storing and reading models in a more machine-readable manner (as opposed to the human-readable textual) languages is needed, also because a machine-readable manner typically provides better runtime performance (as long as no human shall modify the models in this format. EASy-Producer provides a mechanism for IVML, VIL and VTL, implemented as an extension of the respective model management classes. The model persistence mechanism is in `net.ssehub.easy.instantiation.serializer.xml` and depends on `net.ssehub.easy.varModel` as well as `net.ssehub.easy.instantiation.core`.

4.1.6. EASy-Producer Core / Product Line Project Management

So far, loading models is a manual task involving the respective model management mechanisms. The next layer depends on all layers discussed so far and aims at easing the way that EASy-Producer models are loaded and product line code is handles. This happens through the so-called Product Line Project (PLP), which is implemented in an Eclipse-independent manner in `net.ssehub.easy.producer.core`. This bundle also contains the IVML/VIL/VTL default libraries.

4.1.7. EASy-Producer Eclipse Integration

This layer (`net.ssehub.easy.producer.eclipse`) leverages the functionality of all components discussed so far to the Eclipse level, i.e., it scans an Eclipse workspace for PLPs, provides reasoning and VIL execution tasks, etc. However, this involves strong dependencies into Eclipse as well as the plugins, languages and builders used for the respective projects. Running this layer in standalone mode is typically not possible, as frequently opening a workspace fails due to missing dependencies. Fortunately, this is not required for real standalone applications.

4.1.8. EASy-Producer Eclipse UI

We consider the user interface as an independent layer, because a different user interface may present the EASy meachanisms in a target-user specific manner, e.g., for developers or for consultants. The current user interface is implemented in `net.ssehub.easy.producer.ui`, which has dependencies to the EASy-Producer Eclipse integration (and all transitively dependent layers) and to the user interface editors of the EASy-Producer DSL languages (IVML, VIL, VTL). Typically, reusing this layer from outside of Eclipse is not possible at all. However, reusing it within Eclipse works, as shown for QM-IConf, the QualiMater Infrastructure Configuration tool, which utilizes EASy-Producer and even parts of its user interface to enable a graphical configuration of Big Data streaming pipelines.

4.2. Re-using EASy-Producer components within Eclipse

Within Eclipse, the desired components can just be listed in a plugin Manifest. The related extension components will be loaded through OSGi descriptive services as soon as the referencing application will start, i.e., even before any bundle activiators. In this section, we briefly explain how to utilize the models and the DSL language to perform some standalone tasks with EASy-Producer.

Before working with models, we must load them. This happens through the model management classes, i.e., we first tell the three classes where the respective model files are located. Please note, the actual location is in your responsibilitiy, i.e., on this layer it is even possible to you store

and load IVML, VIL and VTL files in totally separated folders. For loading an IVML model, we tell the model management where the IVML files are:

```
VarModel.INSTANCE.locations().addLocation(path,
    ProgressObserver.NO_OBSERVER);
```

Similarly, for `BuildModel` and `TemplateModel` if needed. Please note that adding a location leads to scanning it for models using the respective parser, i.e., the model management indexes the available models for later linking.

Loading a model happens through requesting a descriptor for it. This happens through the functions of

```
VarModel.INSTANCE.availableModels()
```

Various functions are available, e.g., for returning all known models with the same name, for models having a certain version number or even for models located within a certain base URI. Most functions return a set of descriptors, only some functions where all required information to uniqely resolve a model return a single descriptor. For example, let's assume we have name, version and URI available, the we can call

```
ModelInfo<Project> info =  VarModel.INSTANCE.availableModels().
    getModelInfo(name, ver, uri);
```

Obtaining a model may involve loading it (if it is not already cached) and resolving its dependencies. Thus, loading a model may take a little while or even cause exceptions. If already loaded, the `info` containes a resolved reference to the project. If not, one can call the load method of the model management instance. If already loaded, also

```
Project project =  VarModel.INSTANCE.availableModels().
    getModel (name, ver, uri);
```

returns the model.

Having the desired model at hands (we just assume it is `project`), the next step is to create a configuration instance. This happens through

```
Configuration cfg = new Configuration(project);
```

Thereby, a configuration takes over the user decisions made for default and concrete values in the model. This may happen through the default model initialization, or, if a reasoner with respective capabilities is present, through the registered reasoner. The result is an instance containing top-level variables with individual basic or complex values representing the configuration. However, please note that not all values[17] can be changed as you may intend it, as, depending on the underlying model, variables may be frozen.

[17] IVML values are instances with type and must be created through the `ValueFactory` of the IVML model implementation. This requires having the IVML type at hands as well as the respective Java object(s) representing the actual value.

If a dependency to the Product Line Management is feasible, all these steps can be delecated to EASy-Producer by creating a `Persistencer` and passing in an Eclipse project folder. The Persistencer will determine the top-level models according to EASy conventions, create a configuration and provide access to the respective instances. However, this requires that the model has been created with EASy-Producer and specific descriptor files are available.

The configuration can now be used for reasoning. A reasoner shall provide three basic operations:

- *Checking* whether all constraints are fulfilled without changing the configuration.
- *Evaluating* additional constraints on top of a given model without changing the configuration
- *Propagating* values by iteratively applying those constraints that lead to value changes. This operation changes the configuration.

All reasoning operations can be configured, observed in progress and return a result instance indicating whether the operation was successful or which problems have occurred including rather detailed information on the failing constraints, constraint expressions, involved variables etc.

Assuming that a reasoner is available, we perform a propagation[18] with a timeout of 2000 ms and no progress observation through the `ReasonerFrontend`:

```
ReasonerConfiguration rCfg = new ReasonerConfiguration();

rCfg.setTimeout(2000);

ReasoningResult rResult = ReasonerFrontend.getInstance().propagate(
cfg, rCfg, ProgressObserver.NO_OBSERVER);
```

First indication of success is to check `rResult.hasConflict()`.

For executing a given VIL script for this model (assuming we have loaded/obained/stored it in `script`) and performing a self-instantiation into the same `folder`, we call the VIL executor:

```
Executor exec = new Executor(script);

exec.addSource(folder).addConfiguration(cfg).addTarget(folder);

exec.execute();
```

Please note that executing a VIL script may lead to executions during the execution. Moreover, there are more detailed ways of determining paths, even through the PLPs, which we do not discuss here here.

[18] Currently, the ReasonerFrontend contains two rather similar reasoning methods for each purpose, i.e., checking, propagation, validation and evaluation. However, reasoning over a sub-project may be a use case. New convenience methods take the configuration and derive the project from the configuration.

Finally, if all operations are performed and the models are not needed anymore, it is advisable to remove the respective locations from the model management instances to free and save resources.

4.3. EASy runtime: Running EASy-Producer outside Eclipse

As discussed in the previous sections, running EASy-producer heavily relies on that the right extensions and parsers register themselves with the right core classes, i.e., perform a kind of dynamic wiring. In an Eclipse instance, Eclipse and its OSGi implementation Equinox take care of this, so that after specifying the (correct) desired dependencies, the developer does not have to worry about the startup.

However, utilizing Eclipse/Equinox in a standalone environment is not always desired. Although we have to provide dependencies to a large set of Eclipse and, if required, xText core classes, not all of them are actually needed for running EASy-Producer in standalone fashion. To release the developer from thinking about the right startup sequence, which may change over time, we decided to develop a small OSGi environment which is able to startup and shutdown the bundles in the correct sequence without requiring a full OSGi implementation. Packaging all the required components is not trivial, so we typically rely on the default standalone packages produced by the continuous integration (following a default, but customizable setup of standalone components). There, two main packages are created, one containing the EASy-Producer components, one the the required Eclipse components (both approx. 30MB JARs). Some further packages containing required Eclipse dependencies In this packaging, startup and shutdown of EASy-Producer is done by the EASy-Loader. In contrast to OSGi, the EASy-Loader relies on a startup sequence determined during packaging based on the dependencies of the packaged components. This startup sequence is stored in a file the EASy package. Running EASy-Producer in standalone mode, thus requires creating a loader (loads the startup sequence file as a resource from the classloader), executing the startup sequence, performing operations as show above, finally shutting down EASy-Producer. For example, using the EASy-Loader, a startup looks as follows:

```
ListLoader loader = new ListLoader();

loader.startup();

// perform model operations

loader.shutdown();
```

The artifacts including sources and JavaDoc can be obtained from our Maven repository. Releases are in Maven Central, nightly development snapshots are in our repository.

```
<repositories>
  <repository>
    <id>sse</id>
    <name>sse</name>
    <url>https://projects.sse.uni-hildesheim.de/qm/maven/</url>
    <layout>default</layout>
  </repository>
</repositories>
```

The Maven components enable you defining your own EASy-producer composition of core components. In essence, you define in your Maven POM the needed components, specify the EASy startup sequence, run the EASy-Loader and call the relevant functions. We will detail these steps in this section. Please note that if you are fine with a pre-built image, applying the approach introduced above is much more convenient, but implies a higher JAR footprint opposed to a Maven repository with potentially more dependencies in this approach. We present first an example[19] for a Maven-based composition, then a table summarizing the relationship between the components, the Maven artifact specifiers and the startup specification.

Let us assume that we aim at using IVML, the IVML parser and the IVML reasoner to load and validate an IVML model. As main components, we need `de.uni_hildesheim.sse.ivml` (which depends on the IVML model and the foundational classes) and the IVML reasoner `net.ssehub.easy.reasoning.sseReasoner` (which depends on the reasoner core, the IVML model and the foundational classes). For starting up the composition, we need the EASy Loader. As mentioned above, we must perform three steps: Creating a Maven POM, specifying the startup sequence, implementing the code. Please note that you do not need Maven Tycho for such standalone applications.

Step 1: The POM

For realizing the composition, we create the following (fragment of a) Maven POM file, here using snapshot versions and the SSE Maven repository (you may also use release versions from Maven central as shown above):

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>guide</groupId>
  <artifactId>myStandalone</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <repositories>
    <repository>
      <id>sse</id>
      <name>sse</name>
      <url>https://projects.sse.uni-hildesheim.de/qm/maven/</url>
      <layout>default</layout>
    </repository>
  </repositories>

  <dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>net.ssehub.easy.runtime</groupId>
            <artifactId>EASy-dependencies</artifactId>
            <version>1.3.0-SNAPSHOT</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
```

[19] An excerpt from the example in https://github.com/SSEHUB/EASyProducer/tree/master/EASy-Standalone/EASy-Standalone-mvn-individual

```
  </dependencyManagement>

  <dependencies>
      <dependency>
          <groupId>net.ssehub.easy</groupId>
          <artifactId>de.uni_hildesheim.sse.ivml</artifactId>
      </dependency>
      <dependency>
          <groupId>net.ssehub.easy </groupId>
          <artifactId>net.ssehub.easy.reasoning.sseReasoner</artifactId>
      </dependency>
      <dependency>
          <groupId>net.ssehub.easy.runtime</groupId>
          <artifactId>net.ssehub.easy.loader</artifactId>
      </dependency>
      <dependency>
          <groupId>commons-io</groupId>
          <artifactId>commons-io</artifactId>
      </dependency>
  </dependencies>
</project>
```

The first part of the POM file declares the artifact of the example, i.e., guide:myStandalone in version 0.0.1-STANDALONE and jar-Packaging. The next part declares the repository to be used, here the SSE repository akin to Section 4.3. Then we include the EASy dependency management artifact, which contains all pre-existing dependent components and their versions as well as all EASy-producer components[20]. Finally, we declare the three dependencies that we intend to compose, namely IVML parser, IVML reasoner and EASy loader. For some reason, although defined in instantiator core, we also need Apache commons IO as explicit dependency (here the version comes through the dependency management).

Step 2: The startup file

The startup file consists of individual lines indicating wehther either an OSGI activator (`ACTIVATOR`) or a descriptive service (`DS`) shall be started (or stopped). Individual components may require both mechanisms, but typically only one is used. A line starts with the startup mechanism name, a colon and the qualified name of the class implementing the mechanism. Note that there is no space. While for the components from the lower layers of EASy-Producer the sequence can typically be arbitrary, upper layers such as the PLP management shall be mentioned towards the end, best as last entry. Mostly, entries stem from the optional components, but also some common components require entries, such as the instantiator core. The startup file (called ".easyStartup" below) for the example looks like:

```
DS:net.ssehub.easy.reasoning.sseReasoner.Reasoner
DS:net.ssehub.easy.instantiation.core.model.BuiltIn
DS:de.uni_hildesheim.sse.IvmlParser
```

[20] Please note that due to the conventions of Maven Tycho that we use since version 1.3.10 for our build processes, the artefact identifier of almost all EASy-Producer components have changed to their OSGI bundle names.

First the reasoner, then the type registrations of the instantiator core and then the IVML parser is started. Also unused lines may be in the startup file, which then may cause warnings produced by the loader – only serious errors such as class loading errors or linking errors after a startup class has been found and call will interrupt the startup process.

Step 3: Write the code

We just start EASy-Producer, register a location for reading the model IVML (folder "model"), read the IVML model based on a known name ("simple") assuming that only one model does exist, i.e., none with different versions / locations, call the reasoner and shut down EASy-Producer. You can can compose the required steps by your own or you rely on the EasyExecutor (available from version 1.3.0-SNAPSHOT). If you go for the latter, the execution can be stated as

```
    final File sequence = new File(".easyStartup");
    ListLoader loader = new ListLoader(sequence);
    loader.startup();
    new EasyExecutor(new File("."), new File("model"), "simple")
        //do further setup in builder style
        .execute();
    loader.shutdown();
```

Using the executor, it is also possible to execute individual steps (setting up, loading the IVML model, reasoning, VIL execution, discarding the setup) if you need to modify, e.g., the configuration. However, you can also assemble the individual steps in terms of basic code as shown below:

```
public static void main(String[] args) throws Exception {
    final ProgressObserver obs = ProgressObserver.NO_OBSERVER;
    final File modelFolder = new File("model");
    final File sequence = new File(".easyStartup");
    ListLoader loader = new ListLoader(sequence);
    loader.startup();
    final VarModel vm = VarModel.INSTANCE;
    vm.locations().addLocation(modelFolder, obs);
    List<ModelInfo<Project>> models = vm.availableModels().getModelInfo("simple");
    if (null != models && !models.isEmpty()) {
        Project prj = VarModel.INSTANCE.load(models.get(0));
        Configuration cfg = new Configuration(prj);
        ReasonerConfiguration rCfg = new ReasonerConfiguration();
        ReasoningResult rResult = ReasonerFrontend.getInstance().propagate(
            prj, cfg, rCfg, obs);
        System.out.println("Reasoning is ok: " + (!rResult.hasConflict()));
    }
    vm.removeLocation(modelFolder, obs);
    loader.shutdown();
}
```

Please note that the EASy-Loader may not recognize the location for the startup file searching the Jar-classpath. Thus, you can explicitly specify the file in the constructor as done above.

If you have an IVML project and a configuration, just some more steps are needed to run VIL, e.g., to do a self-instantiation. These steps are shown below:

```
List<ModelInfo<Script>> vil = BuildModel.INSTANCE.availableModels()
   .getModelInfo("simple");
if (null != vil && !vil.isEmpty()) {
    ModelInfo<Script> vilInfo = vil.get(0);
    Script script = BuildModel.INSTANCE.load(vilInfo);

    TracerFactory.setInstance(ConsoleTracerFactory.INSTANCE);
    new Executor(script)
        .addBase(new File("."))
        .addSource(new File("."))
        .addConfiguration(cfg)
        .addTarget(new File("."))
        .execute();
}
```

The following summarizes the currently available components as discussed in in Section 4.1, the related Maven artifact specification (group:name, currently all only in version 1.2.0-SNAPSHOT) and the required startup entry. Please note that if you want to use VIL, you will have the instantiator core as transitive dependency, requiring its startup line to be mentioned in the startup file.

Component	Maven artifactId, groupID is always net.ssehub.easy	Startup entry
net.ssehub.easy.producer.core	net.ssehub.easy.producer.core	ACTIVATOR:net.ssehub.easy.producer.core.persistence.internal.Activator
net.ssehub.easy.instantiation.serializer.xml	net.ssehub.easy.instantiation.serializer.xml	DS:net.ssehub.easy.instantiation.serializer.xml.Registration
net.ssehub.easy.instantiation.xvcl	net.ssehub.easy.instantiation.xvcl	DS: net.ssehub.easy.instantiation.xvcl.XVCLInstantiator
net.ssehub.easy.instantiation.velocity	net.ssehub.easy.instantiation.velocity	DS:net.ssehub.easy.instantiation.velocity.VelocityInstantiator
net.ssehub.easy.instantiation.maven[21]	net.ssehub.easy.instantiation.maven	ACTIVATOR:net.ssehub.easy.instantiation.maven.Activator
net.ssehub.easy.instantiation.ant	net.ssehub.easy.instantiation.ant	DS: net.ssehub.easy.instantiation.ant.Registration
net.ssehub.easy.instantiation.aspectj	net.ssehub.easy.instantiation.aspectj	DS: net.ssehub.easy.instantiation.aspectj.Registration
net.ssehub.easy.instantiation.java	net.ssehub.easy.instantiation.java	DS:net.ssehub.easy.instantiation.java.Registration
net.ssehub.easy.reasoning.sseReasoner	net.ssehub.easy.reasoning.sseReasoner	DS:net.ssehub.easy.reasoning.sseReasoner.

[21] The Maven instantiator typically must execute Maven as a process/JVM due to file handle closing problems in Maven. Therefore, the contained lib folder must be unpacked and reachable from your implementation. See the documentation of the instantiator for changing the settings of lib folder and call mode, e.g., to switch to direct Java calls which may prevent instantiating the same POM twice.

Component	Maven artifactId, groupID is always net.ssehub.easy	Startup entry
		Reasoner
de.uni_hildesheim.sse.vil.templatelang	de.uni_hildesheim.sse.vil.templatelang	DS:de.uni_hildesheim.sse.vil.templatelang.VtlExpressionParser
de.uni_hildesheim.sse.vil.buildlang	de.uni_hildesheim.sse.vil.buildlang	DS:de.uni_hildesheim.sse.VilExpressionParser
de.uni_hildesheim.sse.ivml	de.uni_hildesheim.sse.ivml	DS:de.uni_hildesheim.sse.IvmlParser
Included through transitive dependencies		
de.uni_hildesheim.sse.vil.expressions	de.uni_hildesheim.sse.vil.expressions	-
net.ssehub.easy.dslCore	net.ssehub.easy.dslCore	-
net.ssehub.easy.instantiation.core	net.ssehub.easy.instantiation.core	DS:net.ssehub.easy.instantiation.core.model.BuiltIn
net.ssehub.easy.reasoning.core	net.ssehub.easy.reasoning.core	-
net.ssehub.easy.varModel	net.ssehub.easy.varModel	-
net.ssehub.easy.basics	net.ssehub.easy.basics	-

5. EASy-Producer Settings

EASy-Producer defines several system properties to allow modifying default behavior without recompiling it. The following table summarizes all system properties.

System property	Description	Default value
easy.logging.file	File to store EASy core log messages into.	
easy.logging.level	EASy core logging level. May be DEBUG, INFO, WARN, ERROR, OFF.	DEBUG
easy.configuration.useAssignmentResolver	Use the simplified assignment resolver instead of using the available reasoners for initializing a configuration.	false
easy.ui.embeddedEditor	Embed the default EASy editors such as IVML or VIL as tab into the general EASy editor. Otherwise, the editors are only available when opening the respective artifact.	false
easy.notInEclipse	Override detection whether EASy is running in Eclipse. Helpful for jUnit-Tests to be started within Eclipse.	
easy.ivml.core.log	Enable additional output while parsing / validating IVML files.	false
easy.vil.core.log	Enable additional output while parsing / validating VIL files.	false
easy.vtl.core.log	Enable additional output while	false

System property	Description	Default value
	parsing / validating VTL files.	
easy.maven.asProcess	Maven instantiator: Run maven as an own process. false is only effective, if the optional bundle net.ssehub.easy.libs.maven is installed.	true
easy.maven.home	Maven instantiator: External installation folder for Maven jar files.	
easy.maven.classpath	Maven instantiator: Classpath of Maven jar file; If given, used with higher precedence than easy.maven.home.	
easy.maven. classpathExclude	Maven instantiator: Java regular expression for files to exclude from the Maven classpath.	
easy.scenario. instantiate	Run the VIL instantiation as part of the EASy scenario tests.	true

Workspace-specific settings are stored within the Eclipse workspace structures, e.g., settings such as the default reasoner. Examples in .metadata/.plugins are:

- de.uni-hildesheim.sse.ivml.ui/dialog_settings.xml: Dialog settings of the IVML configuration dialog.
- de.uni-hildesheim.sse.vil.buildlang.ui/dialog_settings.xml: Dialog settings of the VIL configuration dialog.
- de.uni-hildesheim.sse.vil.expressions.ui/dialog_settings.xml: Dialog settings of the common VIL/VTL configuration dialog parts.
- de.uni-hildesheim.sse.vil.templatelang.ui/dialog_settings.xml: Dialog settings of the VTL configuration dialog.

If you also have the experimental features installed, the following files may show up:

- de.uni-hildesheim.sse.vil.rt.ui/dialog_settings.xml: Dialog settings of the RT-VIL configuration dialog.
- de.uni-hildesheim.sse.vil.rt.instantiatorCore/rtVilSimulator.properties: Settings of the rt-VIL execution simulator. To be re-loaded/stored when the simulator is used.

Project-specific settings are stored within the project.

- Optional .EASyProducer file in the project root folder. Stores the name and the actual location of the EASy folder containing the EASy-Producer model files. Contains a single line with the location. Helpful, when using EASy-Producer in a Maven setting, e.g., to relocate the EASy folder to src/main/easy.
- .EASyProducer file in the EASy folder containing the EASy-Producer model files. XML-File with related product line projects, but also legacy information required by previous versions of EASy-Producer, e.g., pre VIL instantiators, etc.