EASy Variability Instantiation Language: Default Extensions

Preview Version of 03. April 2025

H. Eichelberger, K. Schmid

Software Systems Engineering (SSE)

University of Hildesheim

31141 Hildesheim

Germany

Abstract

Deriving configured products in product line settings requires turning variabilities into artifacts such as source code, configuration files, etc. Frequently, this is done by domain-specific plugins into the product line tool, often called instantiator, which requires deeper-level programming knowledge about the underlying tool infrastructure. In this document we provide a novel approach for variability implementation. We focus on how to implement selected customization and configuration options in a generic way focusing on the specification of the instantiation process. This enables domain engineers to define their specific instantiation process in a declarative way without the need for implementation of specific tool components such as instantiators.

In this document we document the default extensions for VIL and VTL that can be used and installed optionally. These extensions mostly relate to specific target artifact language capabilities.

_____

Version[1]

| 0.98 | 04. June 2018 | Separation from VIL specification. Default values for parameters. |
| 0.99 | 10. December 2021- | Self-unpacking of Maven libraries, maven settings, additional maven arguments and repository options, Docker instantiator extension, Docker upgrade to 3.8.3, LXC instantiator extension |

Document Properties

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

Acknowledgements

---

[1] Underlined features indicate a change in semantics. Please consider the detailed description in the text or in related footnotes.

_____

Table of Contents

Table of Figures

No table of figures entries found.

1 Introduction

The EASy-Producer Variability Instantiation Language (VIL) [4] aims at instantiating arbitrary artifacts based on a product line configuration given in terms of IVML [3] and, thus, allow realizing one particular important outcome of a software product line. In this document, we summarize and document the extensions to the VIL core language, in particular optional artifact-specific language extensions, e.g., specific capabilities for processing Java source code.

In Section 2 we summarize the default extensions to VIL. In Section 3, we discuss hints on how to use the extensions, in particular collected from practical experience.

2   Default Extensions

In addition to the instantiators and artifact types provided as core (built-in) functionality, there are also some default extensions of VIL/VTL that are usually installed along with EASy-Producer. Depending on the application area, individual installations of EASy-Producer may or may not contain these default extensions and, thus, customize VIL/VTL according to the individual needs or even provide alternative implementations. Table 1 relates the extensions to the implementing bundles. In this Section, we will describe the default extensions shown in Table 1 in terms of their providing types or instantiators, respectively.

Extension	Section	Required Bundle(s)
Velocity	2.1	de.uni-hildesheim.sse.easy.instantiator.velocity
Java	2.2	de.uni_hildesheim.sse.easy.instantiator.java
Java Code	2.3	de.uni_hildesheim.sse.easy.instantiator.java
AspectJ	2.5	de.uni_hildesheim.sse.easy.instantiator.aspectJ
XVCL	2.6	de.uni-hildesheim.sse.easy.instantiator.xvcl
ANT	2.7	de.uni_hildesheim.sse.easy.instantiator.ant
Maven	2.8	de.uni_hildesheim.sse.easy.instantiator.maven[2]
Docker	2.9	de.uni_hildesheim.sse.easy.instantiator.docker[3]
LXC	2.10	de.uni_hildesheim.sse.easy.instantiator.lxc
Yaml	2.11	de.uni_hildesheim.sse.easy.instantiator.yaml
Json	2.12	de.uni_hildesheim.sse.easy.instantiator.json

Table 1: VIL/VTL extensions and providing Eclipse bundles.

2.1   Velocity

The velocity bundle integrates Apache Velocity[4] into VIL/VTL.

Types

This extension does not provide additional types.

Instantiators

The Velocity instantiator [2] allows using one of the basic functionalities of EASy through VIL. It provides instantiator calls for individual templates and collections of templates.

- **setOf(FileArtifact) velocity(FileArtifact t, Configuration c)**
  Instantiates the template in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.

---

[2] Requires a proper execution environment, i.e., JAVA_HOME set or executing JRE set to JDK in Eclipse.

[3] Requires an installation of Docker.

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

- **setOf(FileArtifact) velocity(collectionOf(FileArtifact) t, Configuration c)**
  Instantiates the templates in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result in *t*.
- **setOf(FileArtifact) velocity(Path t, Configuration c, Map m)**
  Instantiates the template in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result into *t*. Thereby, the variable names in *c* are replaced for the template processing by the name-name mapping in *m*, e.g., to enable a mapping of variabilities to implementation names on this level.
- **setOf(FileArtifact) velocity(collectionOf(FileArtifact) t, Configuration c, Map m)**
  Instantiates the templates in *t* through the Velocity engine using the frozen values from configuration *c* and produces the result into *t*. Thereby, the variable names in *c* are replaced for the template processing by the name-name mapping in *m*, e.g., to enable a mapping of variabilities to implementation names on this level.

2.2   Java

The Java extension for VIL/VTL provides several Java-specific types as well as instantiators.

Global Settings Keys (above Types)

On order to resolve type bindings the classpath has to be set. This can be done with the help of the JavaSettings.CLASSPATH key constant. Right now three types are supported.

- **setOf(Path)**
  Specify the classpath as a set of Path.
- **setOf(String)**
  Specify the classpath as set of String
- **String**
  Specify the classpath as String

Types

Java File Artifact

The `JavaFileArtifact` is a composite artifact and allows querying fragment artifacts as well as and (simple) manipulation of Java source code artifacts. Please note that the `JavaFileArtifact` represents a Java compilation unit. In addition to the file artifact operations, this artifact defines the following operations:

- **setOf(JavaClass) classes()**
  Returns all classes defined by the *operand* artifact.
- **JavaClass getClassByName(String n) / classByName(String n)**
  Returns the class in *operand* with name *n*. Nothing happens if the class cannot be found.
- **JavaClass getDefaultClass() / defaultClass()**
  Returns the java class in *operand* with the name of the artifact. Nothing happens if the class cannot be found.
- **setOf(JavaImport) imports()**
  Returns all imports defined by the *operand* artifact.
- **JavaPackage javaPackage()**
  Returns the package declaration defined by the *operand* artifact.
- **setOf(JavaQualifiedName) qualifiedNames()**

Returns all qualified names by the *operand* artifact.

- **renamePackages(String o, String n)**
  Renames all packages in *operand* from *o* to *n*. Nothing happens, if *o* cannot be found.

- **renamePackages(Map m)**
  Renames all packages in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.

- **renameImports(String o, String n)**
  Renames all imports in *operand* from *o* to *n*. Nothing happens if *o* cannot be found.

- **renameImports(Map m)**
  Renames all imports in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.

- **renameQualifiedNames(String o, String n)**
  Renames all qualified names in *operand* from *o* to *n*. Nothing happens, if *o* cannot be found.

- **renameQualifiedNames(Map m)**
  Renames all qualified names in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.

- **modifyNamespace(String o, String n)**
  Renames all packages, imports and qualified names in *operand* from *o* to *n*. Nothing happens, if *o* cannot be found.

- **modifyNamespace(Map m)**
  Renames all packages, imports and qualified names in *operand* according to the mapping in *m* whereby the keys denote old names and the values new names. Nothing happens, if old names cannot be found.

- **Boolean hasAnnotation (String a, String f, String v)**
  Determines whether one of the classes, methods or fields in the *operand* defines the specified annotation named *a* with field *f* and value *v*. The annotation name *a* may be given as qualified or unqualified name[5]. The field name *f* may be given as empty string (then the default name "value" is used).

A `JavaFileArtifact` is created for all real file artifacts with file extension `java`.

JavaClass

The `JavaClass` is a built-in fragment artifact, which belongs to the `JavaFileArtifact`. It represents a single class within a Java compilation unit. The `JavaClass` defines the following operations:

- **setOf(JavaAnnotation) annotations()**
  Returns all annotations defined by the *operand*.

- **setOf(JavaAttribute) attribute()**
  Returns all attributes defined by the *operand*.

- **JavaClass getAttributeByName(String n) / attributeByName(String n)**

---

[5] Please note that we parse Java files but do not resolve them so that qualified names are only available if they are explicitly stated in the source code.

Returns the attribute in *operand* with name *n*. Nothing happens if the attribute cannot be found.

- **setOf(JavaField) attributesByAttribute(String a)**
  Returns all fields defined by the *operand* having an annotation with simple name *a*.
- **setOf(JavaField) attributesByAttribute(String a, String f, Any v)**
  Returns all fields defined by the *operand* having an annotation with simple name *a* and field *f* and value *v* (converted to String for comparison).
- **setOf(JavaMethod) methods()**
  Returns all methods defined by the *operand*.
- **setOf(JavaMethod) methodsByAttribute(String a)**
  Returns all methods defined by the *operand* having an annotation with simple name *a*.
- **setOf(JavaMethod) methodsByAttribute(String a, String f, Any v)**
  Returns all methods defined by the *operand* having an annotation with simple name *a* and field *f* and value *v* (converted to String for comparison).
- **setOf(JavaMethod) classes()**
  Returns all classes defined by the *operand* artifact.
- **setOf(JavaQualifiedName) qualifiedNames()**
  Returns all qualified names defined by the operand artifact.
- **String getName()**
  Returns the identifier defined by the operand artifact.
- **deleteStatement(ExpressionEvaluator evaluator)**
  Deletes the statements selected by *evaluator* within the methods of *operand*. Right now only JavaCalls can be deleted. For examples, see Section 3.3.1.
- **deleteMethodWithCalls(ExpressionEvaluator evaluator)**
  Deletes the methods matching *evaluator* within *operand* and all java calls pointing to this method. For examples, see Section 3.3.1.
- **deleteMethodWithCalls(ExpressionEvaluator evaluator, Any replacement)**
  Deletes the methods selected by *evaluator* within *operand* and all java calls assigned to this method with a replacement that should be inserted for when a method is called. For examples, see Section 3.3.1.

JavaAnnotation

The `JavaAnnotation` is a built-in fragment artifact, which represents a Java annotation attached to a class, a method or an attribute. The `JavaAnnotation` defines the following operations:

- **String getQualifiedName() / qualifiedName()**
  Returns the qualified name of the *operand*. **getName() / name()** returns the simple name.
- **setOf(String) fields ()**
  Returns the fields of the *operand*.
- **String getAnnotationValue(String f)**
  Returns the value of the annotation for field *f* of the *operand*. If the field *f* does not exist, VIL / VTL will ignore the related expression.

JavaMethod

The `JavaMethod` is a built-in fragment artifact, which represents a Java method as part of a class. The `JavaMethod` defines the following operations:

- **String getQualifiedName() / qualifiedName()**
  Returns the qualified name of the *operand*. **getName() / name()** returns the simple name.

- **deleteStatement(Expression e)**

  Deletes all the statements in *operand* that are selected by the expression *e*. Currently, *e* can be of type JavaCall (see below).

- **setOf(JavaAnnotation) annotations()**
  Returns all annotations defined by the *operand*.

JavaCall

A JavaCall is a built-in temporary fragment artifact which is provided by JavaMethod for iterator expressions. A JavaCall defines the following operations:

- **String getType() / type()**
  Returns the (simple name of the) type the *operand* applies to.

JavaAttribute

The `JavaAttribute` is a built-in fragment artifact, which represents a Java attribute (field) as part of a class. The `JavaAttribute` defines the following operations:

- **String getQualifiedName() / qualifiedName()**
  Returns the qualified name of the *operand*. **getName() / name()** returns the simple name.

- **setValue(Any value)**
  Defines the (initial) value of the operand. The value will be emitted as part of the attribute declaration.

- **makeConstant()**
  Turns the *operand* into a "constant" (if it is not already a "constant"), i.e., makes it static final.

- **makeVariable()**
  Turns the *operand* into a "variable" (if it is not already a "variable"), i.e., removes static final.

- **setOf(JavaAnnotation) annotations()**
  Returns all annotations defined by the *operand*.

JavaImport

The `JavaImport` is a build-in fragment artifact, which represents a Java import as part of a class. The `JavaImport` defines the following operations:

- **String getName()**
  Returns the name of the *operand* import declaration.

- **String rename(String n)**
  Sets the n as name of the import *operand*.

JavaPackage

The `JavaPackage` is a build-in fragment artifact, which represents a Java package as part of a class. The `JavaPackage` defines the following operations:

- **String getName()**
  Returns the name of the package declaration in *operand*.
- **String rename(String n)**
  Sets the name of the package in *operand* to *n*.

JavaQualifiedName

The `JavaQualifiedName` is a build-in fragment artifact, which represents a Java package as part of a class. The `JavaQualifiedName` defines the following operations:

- **String getName()**
  Returns the name of the qualified name in *operand*.
- **String rename(String name)**
  Sets the qualified name of *operand*.

Instantiators
Java Compiler

The Java compiler blackbox instantiator allows to directly compile[6] Java source code artifacts from VIL. It provides the following instantiator call

- **setOf(FileArtifact) javac(Path s, Path t, ...)**
  Compiles the artifacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. Additional parameters of the Java compiler can directly be given as named attributes, such as a collection of Strings or Paths or Artifacts denoting the classpath, for example

  ```
  sequenceOf(String) cp = {"$source/lib/myLib.jar"};

  Javac("$source/**/*.java", "$target/bin", classpath=cp);
  ```

- **setOf(FileArtifact) javac(collectionOf(FileArtifact) s, Path t, ...)**
  Compiles the artifacts denoted by *s* into the target path *t*. Additional parameters of the Java compiler can directly be given as named attributes, such as a collection of Strings or Paths or Artifacts denoting the classpath.

JAR File Instantiator

The JAR (Java Archive) file blackbox instantiator allows packing and unpacking JAR files.

- **setOf(FileArtifact) jar(Path b, collectionOf(FileArtifact) a, Path j)**
  Packs the artifacts in *a* into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting JAR file. If *j* does not exist, a new artifact is created with an empty manifest. If *j* exists, the contents of *j* (including the manifest) is taken over into the result JAR, i.e., the artifacts in *a* are added to *j*.
- **setOf(FileArtifact) jar(Path b, Path a, Path j, Path m = null)**
  Packs the artifacts denoted by the path *a* (possibly a path pattern) into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting JAR file. If *j* does not exist, a new artifact is created and *m* (if given) will be the manifest. If *j* exists, the contents of *j* is

---

[6] EASy-Producer must be executed within a JDK so that Java has access to it's internal compiler. A JRE is not sufficient!

taken over into the result JAR, i.e., the artifacts in *a* are added to *j* while the manifest *m* will take precedence over the existing manifest.

- **setOf(FileArtifact) jar(Path b, collectionOf(FileArtifact) a, Path j, Path m)**
  Packs the artifacts in *a* into the JAR file denoted by path *j* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting JAR file. If *j* does not exist, a new artifact is created. If *j* exists, the contents of *j* is taken over into the result JAR, i.e., the artifacts in *a* are added to *j* while the manifest *m* will take precedence over the existing manifest.

- **setOf(FileArtifact) unjar(Path j, Path t)**
  Unpacks artifacts in the JAR file *j* into the target path *t*.

- **setOf(FileArtifact) unjar(Path j, Path t, String p)**
  Unpacks those artifacts in the JAR file *j* matching the ANT pattern *p* into the target path *t*.

- **setOf(FileArtifact) unjar(Path j, Path t, Boolean m)**
  Unpacks artifacts in the JAR file *j* into the target path *t* whereby *m* determines whether the manifest shall also be unpacked.

- **setOf(FileArtifact) unjar(Path j, Path t, String p, Boolean m)**
  Unpacks those artifacts in the JAR file j matching the ANT pattern p into the target path t whereby m determines whether the manifest shall also be unpacked.

2.3   Java Code

The Java extension for VIL/VTL also provides a special artefact type to support code creation and formatting in any sequence (in contrast to the Java artefact/instantiator discussed in Section 2.2). This artefact is currently in development and will be documented as soon as possible.

2.4   Python Code

The Python extension for VIL/VTL also provides a special artefact type to support code creation and formatting in any sequence (akin to the Java Code artifact Section 2.3). This artefact is currently in development and will be documented as soon as possible.

2.5   AspectJ

The AspectJ [1] extension allows to directly compile[6] Java and AspectJ source artifacts from VIL.

Types

This extension does not provide additional types.

Instantiator

The AspectJ extension provides the following instantiator calls

- **setOf(FileArtifact) aspectJ(Path s, Path t, ...)**
  Compiles the artifacts denoted by the source path *s* (possibly a path pattern) into the target path *t*. If *s* is a folder, then all Java and aspect files will be compiled. Additional parameters of the AspectJ compiler can directly be given as named attributes as described above for the Java compiler blackbox instantiator.

- **setOf(FileArtifact) aspectJ(collectionOf(FileArtifact) s, Path t, ...)**

Compiles the artifacts denoted by *s* into the target path *t*. Additional parameters of the AspectJ compiler can directly be given as named attributes as described above for the Java compiler blackbox instantiator.

2.6   XVCL

The XVCL extension allows the usage of the XML-based Variant Configuration Language (XVCL)[7] in VIL scripts. However, this is the integration of a 3rd party research prototype, which is not longer maintained. This instantiator is only for demonstration purpose and contains some known bugs (see below).

Types

This extension does not provide additional types.

Instantiators

The XVCL instantiator integrates XVCL into VIL/VTL. Two different kinds of artifacts are needed to use XVCL, which are usually saved as *.xvcl files:

1) XVCL artifacts describe the instantiation of (code) artifacts, based on XML.
2) A specification, which serves as starting point for the instantiation. This file contains the information about source and destination files, the value settings for the transformation, and a list of files which shall be used for the transformation. We suggest the usage of a template file [4] for the creation of a product specific specification file.

- **xvcl(FileArtifact s)**
  Uses the *s* for instantiation files with XVCL. This file must be structured as follows:

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "null">
<x-frame name="<name of the whole process (will not be used)>"
    outdir="<output destination folder>">
  <set var="<xvcl variable name>" value="<xvcl variable value>" />
  <!-- further variable settings -->
  <set var="dir" value="<sources dir of xvcl files>"/>
  <set var="dtd" value="null"/>
  <set var="out" value="<output destination folder>"/>
  <adapt x-frame="?@dir?<relative path of first file to instantiate>"
    outfile="<destination>"/>
  <!-- further files to instantiate -->
</x-frame>
```

Example

```
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "null">
<x-frame name="project_foo" outdir="C:/Workspace/Project_target/src">
  <set var="boolVar" value="true" />
  <set var="stringVar" value="bar" />
  <set var="intVar" value="42" />
  <set var="emptySetVar" value="{}" />
  <set var="dir" value=" C:/Workspace/Project_source/xvcl_sources"/>
  <set var="dtd" value="null"/>
```

---

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

```
  <set var="out" value=" C:/Workspace/Project_target/src"/>
  <adapt x-frame="?@dir?\main\Main.xvcl" outfile="
    C:/Workspace/Project_target/src/main/Main.java"/>
</x-frame>
```

Known Bugs

- After the specification file was passed to XVCL, it is not possible to delete these file via Eclipse/EASy. It seems that XVCL does not release this resource. Hence, Eclipse/EASy must be closed before this file can be deleted or overwritten.
- XVCL is calling `System.exit(1)` in some cases. We try to avoid such circumstances, but we cannot guarantee that we covered all possible cases. In such a situation, XVCL would close the whole Java Virtual Machine, which will also stop the execution of Eclipse/EASy.

2.7   ANT / Make

The ANT extension allows the execution of ANT build processes[8] (version 1.9.4) as well as Make scripts[9] (through ANT) from VIL.

Types

This extension does not provide additional types.

Instantiators

- **setOf(FileArtifact) ant(Path r, String b, String t)**
  Executes ANT on the build file *b* within path *r* with target *t*.
- **setOf(FileArtifact) make(Path r, String m, String t)**
  Executes MAKE on the makefile *m* within path *r* with target *t*. This instantiator takes arbitrary named arguments that are passed to MAKE.
  This instantiator needs that MAKE is installed on the executing computer.

2.8   Maven

The Maven extension allows the usage of Maven 3.9.7 build processes in VIL scripts. It considers the (inofficial) environment variables MAVEN_SETTINGS_PATH, the settings.xml to be used as user configuration as well as MAVEN_ARGS, an environment variable containing additional arguments to be passed at the end of the command line to Maven.

Types

This extension does not provide additional types[10].

Instantiators

- **maven(Path r)**
  Executes maven with the pom.xml file in *r* with clean and install targets.
- **maven(Path r, Boolean u)**

---

[8] The execution of the Java compiler as part of an ANT build-target requires the "fork"-parameter to be defined as "true", e.g. "<javac fork="true" [...]". If this parameter is not set, the Ant blackbox instantiator may complain about "JAVA_HOME" not being set to the JDK-directory even if this variable is set correctly.

[9] This instantiator is still in development, in particular regarding the installation requirements for non-Unix systems.

[10] Currently, this extension does not support Maven parameters. This will follow in a future version.

Executes maven with the pom.xml file in *r* with clean and install targets and updates snapshots according to *u*.

- **maven(Path r, sequenceOf(String) a, Boolean u)**
  Executes maven with the pom.xml file in *r* updating snapshots according to *u* with additional maven arguments in *a* (only one out of "-U" and *u* is considered, shall not contain "-s" or a local repository property).

- **maven(Path r, String b)**
  Executes maven with the pom.xml file in *b* within *r* with clean and install targets.

- **maven(Path r, String b, Boolean u)**
  Executes maven with the pom.xml file in *b* within r updating snapshots according to *u* with clean and install targets.

- **maven(Path r, String b, sequenceOf(String) a, Boolean u)**
  Executes maven with the pom.xml file in *b* within *r* updating snapshots according to *u* with additional maven arguments in *a* (only one out of "-U" and *u* is considered, shall not contain "-s" or a local repository property).

- **maven(Path r, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *r* with given targets *t*.

- **maven(Path r, Boolean u, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *r* updating snapshots according to *u* with given targets *t*.

- **maven(Path r, sequenceOf(String) a, Boolean u, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *r* updating snapshots according to *u* with given targets *t*. Applies the additional maven arguments in *a* (only one out of "-U" and *u* is considered, shall not contain "-s" or a local repository property).

- **maven(Path r, String b, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *b* within *r* with given targets *t*.

- **maven(Path r, String b, Boolean u, sequenceOf(String) t)**
  Executes maven with the pom.xml file in *b* within *r* updating snapshots according to *u* with given targets *t*

Please note that further maven command line arguments may be passed in through the target list *t*. The VIL maven integration supports several Java system properties, namely

- `easy.maven.asProcess`, a Boolean value (`true|false`) indicating whether Maven shall be executed in a separate process or directly within the EASy producer process. While executing Maven within EASy producer is easier as it is just a method call, the process variant is advisable as Maven otherwise keeps a hold on certain created files (file descriptor leaks described in the Maven issue tracker).
- easy.maven.classpath, the optional full classpath of all Maven libraries if anyway installed externally. This takes precedence over `easy.maven.home` below and implies `easy.maven.asProcess=true`.
- `easy.maven.home`, the optional home folder of a local Maven installation (not set by default, must contain a `lib` folder with all maven libraries, in particular the Maven embedder). Although EASy ships with all necessary binaries for maven, this setting can be helpful in standalone execution of EASy-Producer in order to simplify the installation. This implies `easy.maven.asProcess=true`.

- If neither `easy.maven.classpath` nor `easy.maven.home` is given and Maven shall be executed as a process, the instantiator will try to unpack the included Maven library into a temporary folder and use that folder as `easy.maven.home`.
- `easy.maven.settings` the optional Maven user settings file path, which is typically located in a parent folder of the local repository in the users home directory
- `easy.maven.repo.local` the optional local Maven repository file path, which is typically located in the users home directory

2.9   Docker

The Docker extension allows the usage of Docker, in particular for creating container images. Docker must be installed on the executing/target host for successful execution.

Types

This extension does not provide additional types[11].

Instantiators

- **dockerLogin(String r, String u, String p)**
  Performs a trial login into registry with address *r* using user name *u* and password *p*. If successful, stores the authentication information for registry *r*. Returns true if successful.
- **dockerLoginByToken(String r, String u, String t)**
  Experimental: Performs a trial login into registry with address *r* using user name *u* and identity token *t*. If successful, stores the authentication information for registry *r*. Returns true if successful.
- **dockerLogout()**
  Performs a "logout" by removing all known login information (added before by the login operations). Always returns true.
- **dockerLogout(String r)**
  Performs a "logout" by removing the known login information for registry *r*. Returns true if the information was removed.
- **dockerBuildImage (Path b, Path d, String n)**
  Creates a docker container image from the base/work directory *d* containing all files for the container build context, the docker file *d* and the name *n*. The image name can be given as String either in form "*registry*/*repository*:*tag*" or as "*registry*/*repository*". In the second form, the tag is per default set to "*latest*". The function takes authenticated repositories into account as sources for containers/layers. The function returns the id of the created container. The instantiation will stop if a build error occurs.
- **dockerImageName(String i)**
  Returns the of the image in form "*repository:tag*" for the image with id *I* (Docker does not accept capital characters). The result is undefined if the denoted image does not exist.
- **dockerSaveImage(String n, Path p)**
  Stores the Docker image with name *n* in form "*repository:tag*" to the file in path *p* (in tar format). The result is the set of created image files (usually one).
- **dockerLoadImage(Path p)**

---

[11] Currently, this extension does not support Maven parameters. This will follow in a future version.

Loads the Docker image from path *p* (in tar format).

- **dockerRemoveImage(String i)**
  Removes the image with given id *i* (Docker does not accept capital characters). The result is true if the image was removed. The instantiation will stop if an error occurs.

- **dockerPushImage(String i, String g, String r, String t)**
  Pushes the docker image with image name *i* to repository *r* in registry *g* tagged as *t*. If the image name *i* is not the same as "*registry/repository*", the name will be retagged. For example if we want to push an image into the registry "registry1" and the repository "repository1" with a tag "v1", the image "testapp:01" will be retagged to "registry1/repository1:v1". May require a successful login into *r* before. Waits until completion. Returns true if successful.

Naming conventions

A Docker registry may contain one or more repositories. A Docker registry can be run locally with, e.g., image "registry:2". DockerHub is also a Docker registry. Usually a repository would contain one or more version of the same image type. A tag indicates the version of an image in a repository. An example of this naming convention can be found in DockerHub:

```
registry:       https://hub.docker.com
repository:     python
tag:            3.7-alpine
```

System properties

The VIL Docker integration supports the following Java system properties, namely

- `easy.docker.host`, the host name specification to connect to the Docker process. The default for Windows is "http://localhost:2375", the default for all other operating systems "unix:///var/run/docker.sock".

- `easy.docker.failOnError`: Whether Docker execution errors, e.g., Docker not installed shall lead to a VIL/VTL execution layer. The default value is `true`. For testing, it may make sense to disable errors (value `false`) and to rely on undefined results of the instantiators.

2.10   LXC

The LXC extension allows the usage of LXC, in particular for creating container images. LXC and distrobuilder must be installed on the executing/target host for successful installation.

Types

This extension does not provide additional types.

Instantiators

- **lxcLogin (String r)**
  Sets the systems properties that correspond to two files, the client.crt and client.key. Where r is the path to the directory of both files which is needed for the API to create a connection and perform operations.

- **lxcBuildImage (Path r, Path t, String u)**
  Creates a LXC container image in the base directory r. Directory t points to the template that is going to be used to create the image. On creation the name of the image can be set with u. Because LXC alone cannot create images from a template

file distrobuilder is used to do that. Attention, at the moment distrobuilder command need sudo permissions to execute. If no permission is given to the user the lxcBuildImage instantiator might fail.

- **lxcImageName (String r)**
Returns the name of the image for the image with fingerprint r. The result is undefined if the image doesn't exist or cannot be found.

- **lxcSaveImage (String t, Path r, String u)**
Stores the image with name t in form of a unified tarball (tar.xz) in the defined path r. Using u, a name is assigned to the created file. If the image was created from a container before this will result in a single file otherwise two files (split image) can also occur.

- **lxcLoadImage (Path r, String t)**
Imports the image from a file in path r into the local repository with name set in t.

- **lxcRemoveImage (String r)**
Removes an image from the local repository by the given fingerprint r. Returns true if the image was removed successfully.

- **lxcPushImage (String r, String t, String u)**
Pushes the image with name r to a remote repository t and assigns a new name with u. If u is empty the image will not get a name but will still be imported with a fingerprint. But will be harder to find because the fingerprint is not returned. Before lxcPushImage can be executed, the remote repository t needs to be trusted by client and host manually with `lxc remote add <servername> <ip-address:port>`.

Naming conventions

A LXC repository can either be public or private. But currently there are only a couple official public repositories that can be used to pull images from. Private remote repositories can be set up and used manually. There is no equivalent to DockerHub. A repository most of the time contains multiple different images of a variety of Linux distributions. The syntax to pull from official images repository is:

```
images:ubuntu/jammy
```

Where images is the name of the repository, ubuntu is the distribution and jammy is the release. All available images on that repository can be found on Canonical[12].

For local or private remote repositories replace images with local or the name given when `lxc remote add` was executed. And then use just the image name after the colon.

Systems properties

The VIL LXC integration supports the following Java system properties, namely

- `easy.lxc.host`: the host name specification to connect to the LXC process. The default for linux is "localhost:8443".
- `easy.lxc.failOnError`: Whether LXC execution errors, e.g., LXC not installed shall lead to a VIL/VTL execution layer. The default value is true. For testing, it may make sense to disable errors (value false) and to rely on undefined results of the instantiators.

---

[12] https://uk.lxd.images.canonical.com/images/

2.11 Yaml

The Yaml extension allows for creating/manipulating Yaml files through the `YamlFileArtifact`. Primarily, the `YamlFileArtifact` is bound to `.yaml` and `.yml` files.

Types

Yaml File Artifact

The `YamlFileArtifact` is a composite artifact for creating and manipulating Yaml files. The `YamlFileArtifact` supports multiple contained documents, akin to Spring configurations with multiple profiles. However, the artefact also works with a single and no document (empty Yaml file).

- **YamlFileArtifact()**
  Creates a new artefact instance not related to a physical file. This instance can be used to conditionally collect Yaml documents and later add them to a physical file.
- **YamlNode addDocument()**
  Adds a new document and returns the related node for further operations.
- **YamlNode addDocument(YamlNode n)**
  Adds the given document.
- **deleteDocument(Integer i)**
  Removes the specified document, with *i* as the 0-based index of the document and getDocumentCount()-1 as maximum value.
- **YamlNode getDocument(Integer i)**
  Returns the specified document, with *i* as the 0-based index of the document and getDocumentCount()-1 as maximum value.
- **Integer getDocumentCount()**
  Returns the number of available/stored documents within the actual file.

A `FileArtifact` as well as a `String` (path) can be converted to a `YamlFileArtifact.`

Yaml Node

The `YamlNode` represents a document as well as a hierarchical layer (object, map) within a Yaml document. Where appropriate, the `YamlNode` returns itself or the created subnode for chaining.

- **YamlNode()**
  Creates a new `YamlNode` for deferred/conditionally adding it to a `YamlFileArtifact`.
- **YamlNode addValue(String f, Any v)**
  Adds the field *f* with the given value *v* to the current node. An existing field will be overwritten. Returns this node.
- **YamlNode addValueNotEmpty(String f, Any v)**
  Adds the field *f* with the given value *v* to the current node, but only if *v* is not considered empty (e.g., an empty string). An existing field will be overwritten. Returns this node.
- **YamlNode addValueOrNone(String f, Any v)**
  Adds the field *f* with the given value *v* to the current node. Changes *v* to the Yaml value null/none if v is not considered empty (e.g., an empty string). An existing field will be overwritten. Returns this node.

- **YamlNode addValues(mapOf(String, Any) v)**
  Adds the values in *v* as individual fields to the actual node. Existing fields will be overwritten. Returns this node.
- **YamlNode addMap(String f, mapOf(String, Any) v)**
  Adds field *f* with map *v* as value. An existing field *f* will be overwritten. Returns this node.
- **YamlNode addMap(String f, sequenceOf(Any) v)**
  Adds field *f* with list *v* as value. An existing field *f* will be overwritten. Returns this node.
- **YamlNode addObject(String f)**
  Adds an object-valued field *f* returning the YamlNode of that object node for further operations. An existing field *f* will be overwritten.
- **YamlNode addObjects(String p, String s)**
  Adds multiple, nested object-valued fields as specified in path *p* separated by either character in *s* returning the YamlNode of ultimate object node for further operations. An existing field *f* will be overwritten.
- **delete(String f)**
  Removes the given field *f*. If *f* does not exist, nothing will happen.
- **Boolean has(String f)**
  Returns whether the given field *f* is member of the actual YamlNode.
- **Any getValue(String f)**
  Returns the value of field *f*. May fail/be null, use has(String) before.
- **Boolean isList(String f)**
  Returns whether the value of field *f* is a Yaml list/VIL sequence.
- **sequenceOf(Any) getListValue(String f)**
  Returns the value of field *f* as list/sequence. May fail/be null if *f* is not a list, use isList(String) before.
- **Boolean isMap(String f)**
  Returns whether the value of field *f* is a Yaml list/VIL map.
- **mapOf(String, Any) getMapValue(String f)**
  Returns the value of field *f* as list/sequence. May fail/be null if *f* is not an object/map, use isMap(String) before.
- **Boolean isObject(String f)**
  Returns whether the value of field *f* is a Yaml object.
- **YamlList addList(String f)**
  Adds a list-valued field *f* for incremental creation, i.e., returning the YamlList for further operations. An existing field *f* will be overwritten.
- **YamlNode addList(String f, sequenceOf(Any) v)**
  Adds a (complete) list-valued field *f* with given list values in *v* returning the operand for further operations. An existing field *f* will be overwritten.
- **YamlNode addListNotEmpty(String f, sequenceOf(Any) v)**
  Adds a (complete) list-valued field *f* with given list values in *v*, but only if *v* is not empty. Returns the operand for further operations. An existing field *f* will be overwritten.
- **YamlList getList(String f)**
  Returns the value of field *f* as a YamlList. May fail/be null if *f* is not a list, use isList(String) before.

- **YamlNode addMap(String f)**
  Adds a map-valued field *f* for incremental creation, i.e., returning the YamlNode for further operations. An existing field *f* will be overwritten.
- **YamlNode getMap(String f)**
  Returns the value of field *f* as a YamlNode. May fail/be null if *f* is not a list, use isList(String) before.
- **YamlNode getObjectValue(String f)**
  Returns the value of field *f* as a YamlNode representing a map. May fail/be null if *f* is not an object/map, use isMap(String) before.

Yaml List

The YamlList represents an ordered list of Yaml values, including YamlNode instances.

- **YamlList addValue(Any v)**
  Adds the *v* to the operand (list) by appending the value. Returns the operand list.
- **YamlList addValues(sequenceOf(Any) v)**
  Adds all values in *v* to the operand (list) by appending the values. Returns the operand list.
- **YamlNode addObject()**
  Adds an object-value returning the YamlNode of that object for further operations.
- **YamlList addList()**
  Adds a list-value for incremental creation, i.e., returning the YamlList for further operations.
- **YamlNode addMap()**
  Adds a map-value for incremental creation, i.e., returning the YamlNode for further operations.
- **Integer getSize()**
  Returns the size of the operand list.
- **sequenceOf(Any) getListValue()**
  Returns the operand list als IVML/VTL list.

Formatting

The YamlFileArtifact supports the formatting annotation @format. The related formatter is "yaml", however as there is no specific functionality, currently also "plain" works (discouraged). The formatting supports the following properties:

- indentSteps: The number of indentation steps in whitespaces, as fallback the indentation defined by of @indent (default 1).
- profileArg_indicatorIndent: Whitespaces before "-" (default is same as indentSteps – 1, must be smaller than indentSteps). However, an indicatorIndent smaller than 2 may create superfluous line breaks for nested objects[13]. Then, increase the indentSteps at least to 3.
- profileArg_flowStyle: The Yaml flow style, may be FLOW, BlOCK (the default), or AUTO (implementation decides).
- profileArg_scalarStyle: The Yaml scalar style, may be PLAIN (the default), DOUBLE_QUOTED, SINGLE_QUOTED, LITERAL, or FOLDED. For details on the styles, consult the Yaml file format specification.

---

[13] https://stackoverflow.com/questions/52784112/jackson-yaml-serialization-object-arrays-format

- `profileArg_prettyFlow`: Pretty-print the flows (default is `true`).
- `profileArg_sorting`: Sorting style for maps and objects, may be NONE (implementation decides / arbitrary the default), INSERT (the insertion sequence), ALPHA (alphabetically sorted), COLLATOR (default locale language-based alphabetic sorting).

Instantiators

This extension does not provide additional instantiators.

2.12   JSON

The JSON extension allows for creating/manipulating JSON files through the `JsonFileArtifact`. Primarily, the `JsonFileArtifact` is bound to `.json` files, but can be applied also to files of different extension. The node operations within the Json artifact are just for convenience, i.e., if explicit access to the top-level node via `getData()` is not needed.

Types

Json File Artifact

The `JsonFileArtifact` is a composite artifact for creating and manipulating JSON files. The `JsonFileArtifact` supports multiple contained documents, akin to Spring configurations with multiple profiles. However, the artefact also works with a single and no document (empty JSON file).

- **JsonFileArtifact()**
  Creates a new artefact instance not related to a physical file. This instance can be used to conditionally collect Yaml documents and later add them to a physical file.
- **JsonNode getData()**
  Returns the JSON node representing the data.
- **JsonNode addValue(String f, Any v)**
  Adds the field *f* with the given value *v* to the top node of the document, i.e., the result of `getData()`. An existing field will be overwritten. Returns the node of `getData()`.
- **JsonNode addValues(mapOf(String, Any) v)**
  Adds the values in *v* as individual fields to the top node of the document, i.e., the result of `getData()`. Existing fields will be overwritten. Returns the node of `getData()`.
- **JsonNode addMap(String f, mapOf(String, Any) v)**
  Adds field *f* with map *v* as value to the top node of the document, i.e., the result of `getData()`. An existing field *f* will be overwritten. Returns the node of `getData()`.
- **JsonNode addMap(String f, sequenceOf(Any) v)**
  Adds field *f* with list *v* as value to the top node of the document, i.e., the result of `getData()`. An existing field *f* will be overwritten. Returns the node of `getData()`.
- **JsonNode addObject(String f)**
  Adds an object-valued field *f* to the top node of the document, i.e., the result of `getData()`. Returns the created `JsonNode` of that object for further operations. An existing field *f* will be overwritten.
- **delete(String f)**

Removes the given field *f* from the top node of the document, i.e., the result of getData(). If *f* does not exist, nothing will happen.

- **Boolean has(String f)**
  Returns whether the given field *f* is member of the top node of the document, i.e., the result of getData().

A FileArtifact as well as a String (path) can be converted to a YamlFileArtifact.

Json Node

The JsonNode represents the file contents as well as a hierarchical layer (object) within a Json document. Where appropriate, the JsonNode returns itself or the created subnode for chaining.

- **JsonNode()**
  Creates a new JsonNode for deferred/conditionally adding it to a JsonFileArtifact.
- **JsonNode addValue(String f, Any v)**
  Adds the field *f* with the given value *v* to the current node. An existing field will be overwritten. Returns this node.
- **JsonNode addValueNotEmpty(String f, Any v)**
  Adds the field *f* with the given value *v* to the current node, but only if v is not considered empty (e.g., an empty string). An existing field will be overwritten. Returns this node.
- **JsonNode addValues(mapOf(String, Any) v)**
  Adds the values in *v* as individual fields to the actual node. Existing fields will be overwritten. Returns this node.
- **JsonNode addMap(String f, mapOf(String, Any) v)**
  Adds field *f* with map *v* as value. An existing field *f* will be overwritten. Returns this node.
- **JsonNode addMap(String f, sequenceOf(Any) v)**
  Adds field *f* with list *v* as value. An existing field *f* will be overwritten. Returns this node.
- **JsonNode addObject(String f)**
  Adds an object-valued field *f* returning the JsonNode of that object for further operations. An existing field *f* will be overwritten.
- **delete(String f)**
  Removes the given field *f*. If *f* does not exist, nothing will happen.
- **Boolean has(String f)**
  Returns whether the given field *f* is member of the actual JsonNode.
- **Any getValue(String f)**
  Returns the value of field *f*. May fail/be null, use has(String) before.
- **Boolean isList(String f)**
  Returns whether the value of field *f* is a Yaml list/VIL sequence.
- **sequenceOf(Any) getListValue(String f)**
  Returns the value of field *f* as list/sequence. May fail/be null, use isList(String) before.
- **Boolean isMap(String f)**
  Returns whether the value of field *f* is a Yaml list/VIL map.
- **mapOf(String, Any) getMapValue(String f)**

Returns the value of field *f* as list/sequence. May fail/be null, use isMap(String) before.

- **Boolean isObject(String f)**
  Returns whether the value of field *f* is a Yaml object.
- **YamlNode getObjectValue(String f)**
  Returns the value of field *f* as a YamlNode. May fail/be null, use isObject(String) before.

Formatting

The `JsonFileArtifact` supports the formatting annotation `@format`. The related formatter is "json", however as there is no specific functionality, currently also "plain" works (discouraged). The formatting supports the following properties:

- `indentSteps`: The number of indentation steps in whitespaces, as fallback the indentation defined by of `@indent` (default 1).
- `profileArg_prettyPrint`: Pretty-print the output (default is `true`).
- `profileArg_sorting`: Sorting style for maps and objects, may be NONE (implementation decides / arbitrary the default), INSERT (the insertion sequence), ALPHA (alphabetically sorted), COLLATOR (default locale language-based alphabetic sorting).

Currently, no number of whitespace indentation steps is supported.

Instantiators

This extension does not provide additional instantiators.

3   How to ...?

Learning a new language is frequently simplified if examples are provided. This is in particular true for languages which include a rich library of operations. In addition to the illustrating examples shown in the VIL specification [4] and in this document, we will discuss a collection of typical application patterns related to the default VIL extensions in this section. In particular, this section is meant to be a living document, i.e., this section will be extended over time and is not intended to be comprehensive at the moment.

3.1   VIL

3.1.1 Modifying **Java namespaces**

After a copy operation it can occur that some namespaces need modification because the path has changed. Let's assume we have a package called "test" in our `src` folder and we want to copy it into a new package that contains the name of the project. This modification can be achieved by the following code:

```
postCopy(JavaFileArtifact j, String base) = : {
      j.modifyNamespace("test", "${base}.test");
}



doCopy(String base) = "$target/src/$base/**/*.java" :
  "$source/src/**/*.java" {
  copy(FROM, TO);
  postCopy(TO, base);
}
```

3.1.2 Running XVCL

As described in Section 2.6, XVCL needs a specification file to describe the instantiation of *.xvcl files. The best solution of handling such specification files in EASy is to write a template file. Then use a VIL script to instantiate this template and use the instantiated template for running XVCL. This could be done as below (script for self instantiation):

Template of Specification File

```
template XVCLSpecificationTemplate(Configuration config,
  FileArtifact destFile, Project target) {

  def main(Configuration config, FileArtifact destFile,
    Project target) {
    String codeSourceDir = "${target.getPath()}/xvcl_sources";
    String codeDestination = "${target.getPath()}/src";

    // Template for creating the specification file
    // Header
    '<?xml version="1.0"?>'
    '<!DOCTYPE x-frame SYSTEM "null">'
    '<x-frame name="${destFile.name()}"
      outdir="${codeDestination}">'
    // Product specific value settings
    for (DecisionVariable dv : config.variables()) {
```

```
        createVariableAssignment(dv);
    }
    // Footer
    '  <set var="dir" value="${codeSourceDir}"/>'
    '  <set var="dtd" value="null"/>'
    '  <set var="out" value="${codeDestination}"/>'
    '  <adapt x-frame=?@dir?\\main\\Main.xvcl
      outfile="${codeDestination}\\main\\Main.java"/>'
    '</x-frame>'
  }

  def createVariableAssignment(DecisionVariable variable) {
    '  <set var="${variable.name()}"
      value="${variable.getValue()}" />'
  }

}
```

XVCL VIL Script

```
vilScript XVCL_Project (Project source, Configuration config,
  Project target) {

  version v0;

  main(Project source, Configuration config, Project target)
    = : {
    FileArtifact specificationFile =
      "${target.getPath()}/xvcl_sources/0spc.xvcl";
    clean(specificationFile, target);

    vilTemplateProcessor("XVCLSpecificationTemplate", config,
      specificationFile, target=target);
    xvcl(specificationFile);
  }

  clean(FileArtifact specificationFile, Project target) = : {
    specificationFile.delete();
    Path srcPath = "${target.getPath()}/src";
    srcPath.delete();
    srcPath.mkdir();
  }
}
```

3.2 VIL Template Language

In this section we will discuss some patterns for the VIL template language. Currently, we identified only language-independent how-to hints documented in [4].

3.3 All VIL languages

In this section, we summarize some patterns applicable to both languages (in order to avoid repetitions).

3.3.1 How to remove Java calls

Sometimes you may want to remove some java calls (i.e. logging outputs). Let's assume that our application uses the class `EASyLogger` and we want to remove all warning log calls.

Please note that you need to specify the classpath in order for deletions to take place because the type bindings of the related calls need to be resolved properly. Therefore you can set the classpath directly within VIL. Currently, there are three different ways to specify the classpath. You can specify the classpath as string, set of string or set of path as we show below.

Classpath as set of String

```
main(Project source, Configuration conf, Project target) = : {
    setOf(String) classpath = {"$target", "cp", "test"};
    target.setSettings(JavaSettings.CLASSPATH, classpath);
}
```

Classpath as set of Path

```
main(Project source, Configuration conf, Project target) = : {
    Path newClasspath = target.path();
    Path secondClasspath = target.path() + "/thisIsATest";
    setOf(Path) classpath = {newClasspath, secondClasspath};
    target.setSettings(JavaSettings.CLASSPATH, classpath);
}
```

Classpath as String

```
main(Project source, Configuration conf, Project target) = : {
    target.setSettings(JavaSettings.CLASSPATH, "$target");
}
```

Once the classpath is specified you can start modifying your target code. Removing all warning calls as stated above can be archived by the following code.

```
removeWarnings(JavaFileArtifact j) = : {
    JavaClass cls = j.defaultClass();
    cls->deleteStatement(JavaCall c | c.getType() ==
    "EASyLogger" and c.getName() == "warn");
}
```

You can even go further and delete whole methods that you don't want to use. All assigned calls to this method will be removed as well. Let's assume you want to delete a method called "myMethod" which has no return type. This modification can be archived by the following code:

```
removeMethod(JavaFileArtifact f) = : {
    JavaClass cls = f.defaultClass();
    cls->deleteMethodWithCalls(JavaMethod c | c.getName() ==
    "myMethod");
}
```

If you want to delete a method that has a return type you can add a replacement which will inserted at all java calls. If we assume that the method "myMethod" returns a String the code will be the following:

```
removeMethod(JavaFileArtifact f) = : {
    String message = "Method has been deleted";
```

```
    JavaClass cls = f.defaultClass();
    cls->deleteMethodWithCalls(JavaMethod c | c.getName() ==
    "myMethod", message);
}
```

References

[1]   Project homepage AspectJ, 2011. Online available at: http://www.eclipse.org/aspectj/.

[2]   INDENICA Consortium. Deliverable D2.4.1: Variability Engineering Tool (interim). Technical report, 2012. Available online http://sse.uni-hildesheim.de/indenica

[3]   H. Eichelberger, S. El-Sharkawy, C. Kröher, K. Schmid, IVML language specification. http://projects.sse.uni-hildesheim.de/easy/docs/ivml_spec.pdf [validated: February 2015].

[4]   H. Eichelberger, K. Schmid, EASy Variability Instantiation Language: Language Specification, http://projects.sse.uni-hildesheim.de/easy/docs/vil_spec.pdf [validated: June 2017].