EASy Variability Instantiation Language: Default Extensions

Development Version 0.99 of 18. May 2021

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	18. May 2021	Self-unpacking of Maven libraries

Document Properties

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

Acknowledgements

This work was partially funded by the

- European Commission in the 7th framework programme through the INDENICA project (grant 257483).
- European Commission in the 7th framework programme through the QualiMaster project (grant 619525).
- German Ministry of Economic Affairs and Energy through the ScaleLog project (grant KF2912401SS).
- German Ministry of Economic Affairs and Energy through the IIP-Ecosphere project (grant 01MK20006C).

We would like to thank Aike Sass (Java instantiator, VIL translation-time content expressions, serializer), Bartu Dernek (content assist) and Sebastian Bender (clarifications, ANT and Make instantiator) for their contributions.

[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.java2
AspectJ	0	de.uni_hildesheim.sse.easy.instantiator.aspectJ
XVCL	2.3	de.uni-hildesheim.sse.easy.instantiator.xvcl
ANT	2.4	de.uni_hildesheim.sse.easy.instantiator.ant
Maven	2.4	de.uni_hildesheim.sse.easy.instantiator.maven[2]

Table 1: Default extensions and providing bundles.

2.1 Velocity

The velocity bundle integrates Apache Velocity[3] 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*.
- **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.

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

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

- **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 built-in 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[4]. 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) / attribteByName(String n)**
  Returns the attribute in *operand* with name *n*. Nothing happens if the attribute cannot be found.
- **setOf(JavaMethod) methods()**
  Returns all methods defined by the *operand*.
- **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()**

[4] 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 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[5] 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 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)**

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

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.

AspectJ

The AspectJ [1] extension allows to directly compile[5] 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.

2.3 XVCL

The XVCL extension allows the usage of the XML-based Variant Configuration Language (XVCL)[6] 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)>"
```

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

```
      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"/>
  <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.

2.4 ANT / Make

The ANT extension allows the execution of ANT build processes[7] (version 1.9.4) as well as Make scripts[8] (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)**

[7] 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.

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

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.

2.5 Maven

The Maven extension allows the usage of Maven 3.2.3 build processes in VIL scripts.

Types

This extension does not provide additional types[9].

Instantiators

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

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

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`.

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 [X] 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 2.3, 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].