EASy Variability Instantiation Language: Language Specification

***Preview* version of 11. February 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 define and explain the concepts of the Variability Instantiation Language (VIL) for specifying how customization and configuration options can be turned into (instantiated) artifacts.

Version[1]		
0.5	15. June 2013	first version derived from INDENICA D2.2.2
0.6	8. August 2013	revised concepts
0.7	26. September 2013	revisions based on actual implementation
0.8	21. October 2013	Jars and zips added, clarifications for 'map', RHSMATCH and further built-in operations, qualified names, script parameter sequence refined, pattern path vs. artifact creation clarified, ITER clarified, further XML operations
0.85	30. October 2013	Starting the "How to ...?" section, classloader registration for Java Extensions in VTL
0.86	14. November 2013	Additional XML operations such as constructors, SCRIPTDIR VIL variable
0.87	12. December 2013	Refined hierarchical import path, named-base access to VTL scripts (in addition to file-path based access), final separator expression in for-loop, toJavaPath
0.87	19. December 2013	Java artifacts
0.88	10. February 2014	Project predecessors and further project operations, instantiation via scripts in other projects (instantiate command)
0.89	19. March 2014	Revision of automatic conversions (most specific one is applied).
0.90	04. April 2014	Versions for VTL in VIL scripts.
0.91	21. April 2014	Implementation status, detail fixes, renamePackages in Java Artifact
0.92	23. May 2014 – 02. July 2014	Qualified types, @advice introduces IVML types (Section 3.3.11), null (Section 3.3.9), syntactic additions for the iterated execution in VIL (Section 3.1.9.6), explicit iterator variables instead of ITER (Section 3.3.10)
0.93	03. July 2014 – 12. September 2014	Clarification of external executions in VIL, clarification of load properties, clarification of default extensions and their bundles (document structure changed), EnumValues, "AnyType" renamed to "Any" to avoid conflicts with "Type", random numbers, storing the configuration. Initial additions for instantiating variabilities at runtime (DSPL). Additional functions in Sequence and String. Conversion to sequence for DecisionVariable.
0.94	13. September 2014 – 02. February 2015	Append for sequences, including for sets. Versioned import clarified. Version constraints and Internal version type. Defer declaration removed (cleanup). Rule separating colon is now optional (for convenience). If-statement for VIL. Default extensions for VTL, making the default Random extension transparent. Integer-based sequence creation. `isConfigured` operation for decision variables. Further operations for map. Unmodifiable collections. `map` operation for sequences. Support for nested types. Maven instantiator for VIL. Creation of XML root nodes and how-to on XML. Split operation for String.

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

		JavaPath operations.
0.95		Optional explicit return type for rules (“functions”), setText operation for text representations, clarifications for XMLFileArtifact including a new how-to section, conversions for XMLFileArtifact. Maven instantiator integration improvements, rule refinements. Field notation for @advice, IVML mapping changed, enabling runtime variable changes, separation of the Java default extension into ANT and AspectJ extension. Generic sort function for sequences as well as revert function. Println for debug.
0.96	30. June 2015	Clarification on “Generics”, e.g. Set<Type> changed to setOf(Type), clarification on [] operator.
0.97	10. August 2015-15. March 2016	collect and apply for collections, deleteStatement for Java. FROM/TO instead of RHS/LHS, attributes renamed to annotations according to IVML, asSet/asSequence added as alias operations. protected sub-templates and line end formatting for VTL. Clarification of global settings and deleting java calls, disallow cyclic imports..
0.98	04. June 2018	for/while for VIL, while for VTL, sort for maps, conversion for mapped IVML projects, enableTracing, notifyProgress, OCL/IVML alignment: Real operations, Integer operations, String operations and locale, Type operations including allInstances, getType on all types (change of getType for configuration types), enum literal qualification by ‘::’ as in IVML, collection operations (selectByKind, selectByKind vs. selectByType, sum, product, includesAll, excludesAll, asSet, asSequence, flatten, any, one, exists, forAll, isUnique, sortedBy, sets with difference and symmetricDifference, collect, collectNested, sequences with subsequence, union, prepend, insertAt, hasDuplicates, overlaps, isSubsequenceOf, typeReject), operations for ordered IVML enums. Import of VTL scripts from VIL for reuse. Range comparisons. Default values for parameters. Moved language extensions into own document [16]. Chained content statements, replaced “print” by “!<CR>”, explicit content flush, content-alternative and content-loop. Introduction of XMLnode and XMLcomment as well as related operations (elements operation changed to sequence return type), more XML meta operations. Recursive expression expansion, extended expression expansion for VTL file templates. VIL/VTL compound type. More flexibility in VTL element sequence.
0.99	11. Feburary 2025 -	Clarifications, e.g., for dynamic dispatch. *getOrAdd* operation for Map, VTL formatting hints, setExecutable for file system artifacts. String osName and getProperty. Wildcard imports and experimental inserting of imported functions into actual language units for extensible dynamic dispatch. Syntax of rule annotations (without semantics for now). File artefact operations getMd5Hash and hasSameContent. VTL content formatter. hasConfiguredValue, hasDefinedValue. Builder block expression.

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.

Table of Contents

Table of Figures

1 Introduction

This document specifies the Variability Instantiation Language (VIL) in terms of the most current version of the language.

VIL consists of two languages: a build process description language and a template language. The focus of VIL is on instantiating a whole product line, while the template language focuses on the instantiation, creation or generation of individual artifacts. All VIL languages are based on an explicit and extensible artifact model as well as a tight integration with the Integrated Variability Modelling Language IVML [3].

The remainder of this language specification is structured as follows: in Section 2, we briefly introduce the VIL approach. In Section 3, we define the syntax and semantics of the aforementioned VIL languages, their common expression language as well as the underlying type system (including the artifact model and the IVML integration). Please note that default language extensions, e.g., for Java are since version 0.98 documented in [16]. In Section 4 we provide answers to frequently asked questions in terms of a how-to collection. In Section 5 we discuss the implementation status, in particular known deviations from this language specification. Finally, in Section 6, we provide the grammars of the VIL languages as a reference.

2 The Approach

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

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

- **Artifact meta-model:** Everything that can be instantiated (transformed or generated) is regarded as an artifact. The VIL approach relies on an artifact meta-model as its foundation. The artifact meta-model (or often artifact model for short) describes what operations can be performed on certain types of artifacts, 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 artifacts using the capabilities of the assets for specifying the instantiation.
- **VIL template language** is used to instantiate a certain type of target artifact in a reusable way. Basically, the VIL template language covers generation as well as transformation-based production strategies.
- **Blackbox instantiators:** In some situations it might be difficult, inconvenient, or even impossible to describe a production strategy using the VIL template language. One example is the Cocktail instantiator discussed in Deliverable D2.2.2 [5] as it mainly modifies Java bytecode and, thus, it is easier to realize (at least some part of it) in an usual programming language such as Java, i.e., from the point of view of VIL as a black box. Another example is a programming language compiler or a linker, which should not be re-developed using VIL but simply reused. In case of legacy product lines, an existing instantiator may be called or wrapped into a VIL extension.
- **VIL control language:** This is the main part of VIL as it binds all other pieces together. This is used to define individual production strategies, i.e., to relate artifacts and instantiation mechanisms, to combine production strategies in terms of rules and to specify the execution of the rules. Basically, it is a rule-based programming language as a foundation for describing product line instantiation processes. We will call this language for short VIL.

VIL and its sublanguages are tightly integrated with IVML, i.e., IVML identifiers, configuration values, and types can be directly used in VIL. From a more general point of view, VIL and its sublanguages rely on existing, practically proven concepts such as build rules or template languages to avoid reinventing the wheel. However, existing concepts as well as related tooling does not provide the full support for variability instantiation as we experienced in our analysis of related technologies. Thus, we reuse and extend existing concepts to apply it to variability realization and created the VIL as a completely new language along with a novel implementation.

3 The VIL Languages

In this section, we will describe the two (sub) languages of VIL, i.e., the VIL (control) language and the VIL template language (VTL), as well as their main concepts. While VIL aims at defining the overall instantiation process for a software product line VTL specifically aims at supporting the generation and instantiation of textual artifacts. We describe VIL in Section 3.1 and VTL in Section 3.2. Due to the nature of both (sub) languages, they share a common type system as well as a common expression language. For working with different kinds of artifacts and variabilities, VIL provides an extensible type language. We will describe the expression language in Section 3.3 and the type library in Section 3.3.12. Default extensions for specific programming languages are summarized in [16]. The **extensible VIL type system** is the foundation for both VIL sub languages. The type system consists of basic types such as Integer or Boolean, configuration-related types realizing the integration with an IVML [3] variability model, artifact-related types implementing the artifact (meta) model, implicit types representing instantiators and derived types such as collections (including IVML collections). In particular, the type system is extensible, i.e., additional or refining artifact types or instantiators can easily be added (in terms of Java classes). If compared with an object-oriented language, the artifact types can be considered as classes, the operations as methods, individual artifacts as instances and the execution of artifact operations as method calls. However, the instantiators can be more aptly compared to transformation rules as they are first of all rule-based and second operate on the artifact model, but are themselves not part of it. The **common VIL expression language** represents a wide range of expressions from simple calculations over artifact operation and instantiator calls up to rather complex composite expressions. The expression language relies on the operations and operators provided by the VIL type system.

The two VIL languages are realized on top of the common expression language. Both languages follow a textual approach to the specification of artifact and product instantiation and support batch processing. Our definition of the syntax of VIL draws upon typical concepts used in programming languages, in particular Java, build languages such as *make*, template languages such as *xtend* as well as expressions inspired by IVML and the Object Constraint Language (OCL) [6]. We adapt these concepts as needed to provide additional operations required in variability instantiation, such as the integration with a variability model or an explicit artifact model.

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

- The syntax as well as the examples will be illustrated in `Courier New`.
- **`Keywords`** will be highlighted using bold font.
- *`Elements and expressions`* that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- We will denote optional parts of syntax descriptions by *`[...]`*.

- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number.
- Statements will be separated using semicolon "`;`" (most other language concepts may optionally be ended by a semicolon).
- Different types of brackets will be used to indicate lists "`()`", sets "`{}`", etc. This is closely related to the Java programming language.
- We will indicate comments using "`//`" and "`/* ... */`" (cf. Java).

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

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

In Section 3.1, we will describe the specific concepts of the VIL which is responsible for specifying the overall variability instantiation process of an entire (hierarchical) product line. In Section 3.2, we will describe the concepts of the VIL template language, which provides the means to describe the instantiation of a single (textual) artifact. Basic concepts of the VIL build and the VIL template language are rather similar (also to IVML) in order to simplify learning and application of these languages. In Section 3.3, we will detail the common expression language which is part of both, the VIL build and the VIL template language. In particular, we will detail the type system, i.e., the built-in types, their individual operations and the default instantiators that are part of the VIL implementation. In Section 3.3.12 we discuss the built-in VIL types and operations, i.e., the extensible library of types and operations that can be used to instantiate variability. In addition to the built-in operations, there are several default extensions of VIL, e.g., for Java artifacts [16]. Extensions are optional and may or may not be installed depending on the actual product line setting.

3.1 Variability Instantiation Language

In this section, we describe the concepts and language elements of the VIL (control) language (for short VIL) in detail. This language aims at specifying the variability instantiation process of a whole (hierarchical) product line (as opposed to the instantiation of a specific artifact type covered by the VIL template language).

However, VIL focuses on the implementation and instantiation of variabilities rather than on the entire build process of a whole system. Thus, VIL is intended as an extension to existing build languages, i.e., it shall be integrated with those languages rather than replacing them.

3.1.1 Reserved Keywords

In VIL, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by the keywords of the common VIL expression language given in Section 3.3.1.

- **`@advice`**
- **`const`**
- **`else`**
- **`exclude`**
- **`extends`**
- **`execute`**
- **`for`**
- **`if`**
- **`import`**
- **`insert`**
- **`instantiate`**
- **`join`**
- **`load`**
- **`properties`**
- **`protected`**
- **`requireVTL`**
- **`typedef`**
- **`version`**
- **`vilScript`**
- **`wile`**
- **`with`**

3.1.2 Scripts

In VIL, a script (**`vilScript`**) is the top-level element. This element is mandatory as it identifies the production strategies to be applied to derive an instantiated product. The definition of a script requires a name, as a basis for referring among VIL scripts and a parameter list specifying the expected information from the execution environment such as the actual configuration or the projects to work on. In order to realize the necessary capabilities required for implementing the hierarchical product line capabilities of the EASy producer tool, at least the source project to instantiate from, the target project to be instantiated and the actual variability configuration must be passed to a VIL script.

Basically, VIL may refer to all visible configuration settings in a variability configuration. In traditional product line settings, VIL will refer to the actual values of frozen decision variables (and their underlying structure). In Dynamic Product Line Settings (DSPL) [10, 11], VIL will refer to all variables including unfrozen ones.

In order to make this integration with the variability model explicit, these decision variables may be directly referred in VIL by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. This

may complicate the development of VIL scripts as actually unknown identifiers will at least lead to a warning. To support the domain engineer in specifying valid scripts, VIL provides the **advice** annotation specifying the name for the IVML models used in the VIL scripts. Qualified names resolvable via the **advice** annotation do not lead to warnings in the VIL editor. Further, all types in IVML models (in case of overlaps in terms of their qualified names) are made available. On these types, each declared decision variable is available through an operation named according to the decision variable in order to ease the access. As an explicit version number may be stated for VIL scripts (akin to IMVL models), also advices and model imports may be version-constrained.

Optionally, a VIL script may extend another VIL script, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

Syntax:

```
//imports
@advice(ivmlName)
vilScript name (parameterList) [extends name1] {
 //optional version specification
 //loading of variable values from an external source
 //variable definitions, type defs, compound defs
 //rule declarations
}
```

Description of syntax: the definition of a VIL script consists of the following elements:

- First, all referenced scripts must be imported. We will detail the import syntax in Section 3.1.4.
- Optional advices declaring the underlying variability models.
- The keyword **vilScript** defines that the identifier *name* is defined as a new script with contained production strategies.
- The parameter list denotes the arguments to be passed to a VIL script for execution. When executed by EASy, at least the source project(s), the target project, and the variability configuration must be passed in. Source and target project may be identical in case of (traditional) in-place instantiation. In case of multiple source (predecessor) projects, the source parameter shall be given as a collection. During processing, subsequent predecessors may be accessed via the `Project` type. However, further named parameters may be given upon an explicit invocation from an external call, e.g., an integration with a build language such as ANT or Maven. Parameters may have default values given in syntax *Type name = value* listed after all parameters not declaring default values. If given as first parameters in sequence project-configuration-project, the

parameters will be bound independently of their name and, thus, the parameters can be named arbitrarily. Otherwise (due to the option of named parameters), the sequence of the parameters may be arbitrary but the parameters must exactly be named `source`, `config` and `target`. An implementation shall treat unspecified parameters as undefined, i.e., expressions using this parameter are undefined.

- A VIL script may optionally extend an existing (imported) VIL script. This is expressed by **`extends`** *name*$_1$, whereby *name*$_1$ denotes the name of the script that shall be extended.
- Production strategies are described within the curly brackets.

Example:

```
@advice(YMS)
vilScript YMSBuild(Project source, Configuration config,
        Project target){
  /* Go on with the production strategies for YMS here */
}
```

Please note that the types shown above such as `Project` or `Configuration` will be explained in detail in the next section. Further, a VIL script for multi-product lines may require a collections of projects (see Section 3.1.5.4), while single project parameter is sufficient for a traditional product line build.

A script defines the following two implicit variables:

- `OTHERPROJECTS` is a collection variable which contains the artifacts created by the execution of VIL scripts in other projects.
- `SCRIPTDIR` is a String variable containing the location of this script in the file system.

3.1.3 Version

Akin to IMVL, VIL build specifications may optionally be tagged with an explicit version number in order to support product line evolution. Evolution of software may yield updates to projects, IVML models and VIL scripts so that scripts of different versions may exist and need to be clearly distinguished.

Syntax:

```
// Declaration of the version of a VIL script.
version vNumber.Number;
```

Description of Syntax: A version statement consists of the following elements:

- The **`version`** keyword indicates a version declaration. At maximum one version declaration may be given in a VIL build file at the very first position within a VIL script.
- ***`v`****`Number.Number`* defines the actual version of the project (here only two parts prefixed by a "v"). At least one number must be given and no restriction holds on the amount of sub-version numbers.
- A version statement ends with a semicolon.

Example:

```
vilScript YMSBuild(Project source, Configuration config,
      Project target){
      version v0.1.4;
      ...
}
```

3.1.4 Imports

The production strategies for a variability instantiation build process may be defined in a single VIL script or may be reused from other (existing) scripts. Therefore, VIL scripts may be imported. Cyclic imports are not allowed and shall cause an error. In order to support also the evolution of product line build specifications, VIL allows the specification of version-restricted imports. Imports make the production strategies defined in the specified build file accessible to the importing script.

Syntax[2]:

```
// Unconstrained and constrained imports.
import name;
import name with expression;
```

Description of Syntax: An import of a scripts consists of the following elements:

- Importing a VIL script starts with the keyword **`import`**. Multiple imports may be given in a VIL build file directly at the beginning of the script file.
- *`name`* (given in terms of a VIL identifier) refers to the name of the script to be imported. However, multiple scripts with identical names and versions may exist in a file system, in particular in hierarchical product lines. Thus, imports are determined according to the following **hierarchical import**

[2] Since September 2022, there is an experimental import variant indicated by keyword `insert` rather than `import`. While `import` makes the referenced language concepts available, `insert` virtually inserts the defined functions into the language unit(s) declared in the same file and, thus, allows for extending the dynamic dispatch by those functions. This can be used to realize open/extensible models, in particular in combination with wildcard imports. If there is a basis language unit to base an extension on, it is advisable to import that unit and not to refine/extend it. Currently, no local variables shal be used in the language unit to be inserted.

convention, i.e., starting at the (file) location of the importing script (giving precedence to imports in the same file) the following locations are considered in the given sequence: The same directory, then contained directories (closest directories are preferred) and finally containing directories (also here closest directories are preferred). In addition, sibling folders of the folder containing the importing model and predecessor projects are considered[3]. Similar to Java class paths, additional script paths may be considered in addition to the immediate file hierarchy. The name may end with a * indicating a wildcard import of models with names that start with the prefix name until the *. VIL can import VTL (two default parameters, target not set, created contents is ignored / discarded) and call functions defined there in order to facilitate reuse.

- An optional restriction of the import in terms of an expression. This is indicated by the keyword **`with`**[4]. followed by a parenthesis containing the restrictions. A restriction is stated using the implicit variable **`version`** denoting the version of the import an operation supported by the version type (==, >, <, >=, <=, see Section 3.4.6) and a version number. More complex constraints can be stated using Boolean operations. In case of multiple matching versions, the model with the highest version number is selected by default.
- An import statement ends with a semicolon.

Example:

```
vilScript YMSBuild(Project source, Configuration config,
      Project target){
      version v0.1.4;
      import generics with (version >= v1.12);
}
```

VTL templates may not be imported but restricted with respect to their version. The syntax looks similar to imports, but the syntax is adjusted to the actual way VTL templates are called from VIL. Using the syntax below, VTL scripts with name “name” are restricted to the given version akin to the import of VIL scripts.

```
restrictVTL "name" with expression;
```

[3] Actually, EasY-Producer stores the imported parent product line models in individual subfolders (starting with a “.”), i.e. possibly sibling folders of a model.

[4] Previous versions required parenthesis and supported only a rather limited set of expressions. Since version 0.93, arbitrary expressions can be used here, but currently we support only version expressions.

3.1.5 Types

Basically, VIL is a statically typed language with partially postponed type checking at runtime as we will detail below. Thus, VIL provides a set of formal types to be used in variable declarations or parameter lists. We distinguish between basic types, configuration types, artifact types, and collection types. The respective function declarations (rules in VIL, cf. Section 3.1.9 and sub-templates in VTL, cf. Section 3.2.9) can be used to describe further operations on a basic or user-defined type, e.g., using respective type as the first parameter. Also dynamic dispatch (cf. Section 3.1.9.3) applies.

3.1.5.1 Basic Types

The basic types in VIL correspond to the basic types of IVML, i.e., Boolean (`Boolean`), integer (`Integer`), real (`Real`) and string (`String`) with their usual meaning.

Boolean constants are given in terms of the keywords true and false. Integer constants are stated as usual numbers not containing a "." or an exponential notation. Real constants must contain the floating-point separator "." or may be given in exponential notation. Strings are either given in quotes or in apostrophs and may contain the usual escape sequences including those for line ends, quotes and apostrophs.[5]

3.1.5.2 Configuration Types

A configuration type denotes the representation of IVML configuration elements in VIL. However, due to the nature of VIL, we need only access to the configuration and the structure of an IVML model rather than to all modelling capabilities. Thus, VIL provides a specific set of built-in configuration types. The actual instance of a configuration is passed into a VIL script in terms of a script parameter. Configuration types cannot be directly created in a VIL script and must not modify the underlying IVML model.

The entry point to a configuration in terms of an IVML model is the type `Configuration`. It provides access to all decision variables and annotations, depending on the product line settings frozen variables only or all variables. In particular, `Configuration` allows creating projections of a given configuration in order to simplify further processing. Further, it provides access to IVML type declarations such as compounds or enumerations and their value. This is represented by the `IvmlElement` and its subtypes. An `IvmlElement` represents IVML concepts in a generic way and provides access to its (qualified) name, its (qualified) type name and the configured value. Specific subtypes of `IvmlElement` are `DecisionVariable`, `Annotation` and `IvmlDeclaration`, each providing more specific operations as we will discuss in detail in Section 3.4.6.

[5] Strings delimited by quotes may contain apostrophs, strings delimited by apostrophs may contain quotes.

3.1.5.3 Artifact Types

Artifact types represent the different categories of artifacts used in the artifact model. Some artifact types are built-in and part of the VIL implementation, while further types can be defined in terms of an extension of the artifact model. In this section, we will discuss only the predefined types. Please refer to the EASy developers guide on how to define more specific artifact types (as well as how to integrate instantiators implemented in a programming language).

The type `Project` is a mapping of a physical project (Eclipse) into VIL and provides related operations such as mapping paths between the source and the target project for instantiation.

A `Path` is a predefined type of the VIL artifact model although it is not an artifact by itself. A `Path` represents a relative file system path and may possibly contain wildcards. A path is specified in terms of a `String` in VIL and is automatically converted into a `Path` or an artifact instance depending on the actual use. In more detail, paths are specified according to the ANT [9] conventions, i.e., using the slash as path separator and wildcards for patterns. The following wildcards are supported: `?` for a single character (excluding the path separator), `*` for multiple characters (excluding the path separator) and `**` for (sub) path matches.

`Artifact`[6] is the most common artifact type and root of the VIL artifact hierarchy. The predefined `Artifacts` have also predefined methods. For example, they allow to delete the artifact (if possible at all), or to obtain access to its plain textual or binary representation. VIL provides a set of built-in artifact types such as `FileArtifact` and `FolderArtifact` which are both `FileSystemArtifacts`. Further, VIL provides more specific artifact types such as the `VtlFileArtifact` representing VIL template files (see Section 3.4.8) or the `XmlFileArtifact` representing parsed XML files with a substructure of specialized fragment artifacts such as `XmlElement` or `XmlAttribute`. Please note that artifact instances are assigned in a polymorphic way, i.e., while a `FileArtifact` may be specified as type in a VIL script, it may actually contain a more specific type. However, pattern paths, i.e., paths containing wildcards, will not be turned into artifact instances.

3.1.5.4 Collection Types

VIL provides three collection types, sequences (keyword `sequenceOf`), sets (keyword `setOf`) and associative containers (keyword `mapOf`). Collection types are generic with respect to their content type(s) and, similarly to IVML, the content type must be stated explicitly, such as `sequenceOf(Integer)` or `setOf(DecisionVariable, FileArtifact)`. Akin to IVML, collections can be nested and contain further collections.

Sequences may contain an arbitrary number of elements of a given element type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. In sequences, elements can be accessed by their position in the collection using an index (`[`*`index`*`]`). In VIL, indexes start at zero and run until the

[6] We adopted US English in the implementation of VIL.

number of elements in the collection minus one (as in Java and many other languages). Collections typically occur as results of operations, rule, or instantiator executions. In addition, they can be explicitly initialized using type-compatible expressions of the appropriate dimension as shown follows

```
sequenceOf(Integer) someNumbers = {1, 2, 3, 4, 5};
setOf(Integer, Integer) somePairs = {{1, 2}, {3, 4}};
```

A `Map` represents an associative collection in VIL, i.e., a collection which relates keys to associated values. In particular, it allows retrieving the value assigned to a key via the `get` operation and the `[]`-Operator (`[`*key*`]`). Basically, associative collections are intended to simplify the translation of IVML-identifiers to implementation-specific identifiers in individual artifacts. Therefore, VIL associative collections can be explicitly initialized in terms of key-value-pairs using type-compatible expressions

```
mapOf(String, String) idTranslation
  = {{"nrOfProcessors", "procCnt"}, {"nrOfNodes", "nodeCnt"}};
```

VIL supports a set of operations specific for collection types, e.g., excluding, projecting, or collecting elements in a collection, etc. We will introduce the full set of operations in Section 3.4.5.

3.1.5.5 Compound Types

In many cases, the types introduced so far are sufficient even for complex scripts. However, in some cases it is necessary to work with complex heterogeneous types, in particular types that are not already defined through IVML rather than locally by VIL scripts. For this purpose, VIL allows defining own compound types, similar to IVML compounds or usual programming language records. A compound consists of typed variables called slots, which may have default values. Compound instances can be created by the constructor of the compound type, using named parameters values for individual slots can be assigned. Akin to IVML, compounds can be abstract and refined (although slots cannot be shadowed by redefining them).

Syntax:

```
[abstract] compound name [refines Type] {
    [const] Type name1 [= value];
}
```

Description of Syntax: The declaration of compound consists of the following elements:

- The optional keyword **`abstracts`** identifies a non-instantiable (base) compound.
- The keyword **`compound`** indicates the definition of a complex (heterogeneous) type. *`name`* denotes the name of the new compound.
- The optional keyword **`refines`** indicates that the new compound shall inherit all slots from *`Type`*.
- Within a compound, arbitrary variable (slot) definitions with optional default value can be given. The names of the sloths must be unique within

the defined hierarchy of compound types. Slots can be defined constant and are then read-only.

- A compound definition can optionally end with a semicolon.

Compounds can be instantiated using the compound constructor, i.e., for a compound type named *name* the constructor is called by **new** *name*(); Values for slots can be given as named parameters to the constructor call, whereby type compatibility is checked at runtime. Slots can be accessed using the dot-operator. In case of constant slots, the value of a slot cannot be changed except for initial values.

Example:

```
abstract compound Base {
    Integer a = 5;
    String b;
}

compound Refined refines Base {
    Real r = 1.1;
};
```

Then declarations above define the abstract compound type `Base` with two slots, slot `a` with default value and slot `a` without default. The compound type `Refined` with refines the compound type `Base`, inherits all slots and defines a further slot with default value. While `Base` cannot be instantiated, instances of `Refined` can be created using a constructor. Of course, instances of sub-types can be assigned to supertypes.

```
Refined r = new Refined();
Base b = new Refined();
```

To change the value assignments to slots, named parameters can be used.

```
Refined r1 = new Refined(a = 7, b = "abba", r = 2.2);
```

Field access through the dot-operator can be used for reading and (if not constant slots are accessed) also for writing, e.g., `b.a = 7;` or `r.r = 2.7;` Only slots known by the respective type can be accessed. Of course, compound types can be used in collections, e.g., sets, sequences or maps.

3.1.6 Variables

A variable provides name-based access to a value of a certain type (see Section 3.1.5), similar to variables in programming languages.

In VIL, the value of a variable can be modified at any time (in contrast to build languages such as ANT [9] where a value of a property can be set only once). In addition, a variable may be declared to be constant so that a value can be set only once and not be modified afterwards. Variables may be of global scope, i.e., directly defined within a VIL script or they may be local (within rules, see Section 3.1.5.5).

Syntax:

```
// Declaration of a variable.
Type variableName1;
Type variableName2 = value;
const Type constantName = value;
```

Description of Syntax: The declaration of variables consists of the following elements:

- The ***Type*** defines the type of the variable being declared.
- The identifiers *variableName*$_1$, *variableName*$_2$ and *constantName* are the names of the declared variable or constant, respectively.
- The optional keyword *const* indicates that a variable can be defined only once and the value must not be redefined.
- A variable may optionally be initialized by a value or an expression, which evaluates to a value of the given type.
- Variable declarations end by a semicolon.

Parameters of VIL scripts are declared akin to variables, but without an initial value.

Example:

```
Integer numberOfCompilerProcesses = 4;
sequenceOf(Project) sources;
sequenceOf(Project, DecisionVariable) mapping;
```

Variables may be referred in Strings such as path patterns. A variable reference is stated as `$variableName`. Even entire VIL expressions (see Section 3.3) including variables may be given in Strings in the form `${expression}` and are expanded (if specified in nested/recursive manner) to the actual value of `expression`. The expansion can be quoted/escaped by prefixing a backslash in Java notation, i.e. `\\` prevents the respective replacement and causes that `$variableName` or `${expression}` is emitted as given without the escaping backslash). When applying the respective String, variable, and expression references are substituted with their actual value.

3.1.7 Externally Defined Values of Global Variables

Global variables or constants are defined as part of a VIL script. The value of a global variable or constant may be specified by an external source, e.g., to customize the VIL script according to the build environment (similar to properties in ANT [9]). For externally defined values of variables, initial values are not needed, in particular also not for constants. Externally specified values are subject to automated type conversion and variable reference or VIL variable (not expression) substitution based on the VIL script arguments. Multiple external property files are processed in sequence so that the variable values defined by external files listed before are overwritten (accidental constant redefinition will lead to an execution error).

Syntax:

```
// Loading the values of global variables
load properties "path";
```

Description of Syntax: Loading values from an external file consists of the following elements:

- The keywords ***load properties*** indicates that the values of global variables shall be loaded from an external file. Multiple load statements may be given in a VIL script directly after the version statement.
- The path points to the file containing the initial values of the variables. Relative paths are interpreted relative to the target project of the VIL script, i.e., in case of multiple instantiations all properties must be set in the (partial) target project. Also absolute paths may be given, in particular using variable references as described in Section 3.1.5.5. The file must be given in Java properties format, i.e., each line specifies the value of a specific variable in the following form

  `variableName = value`

  In addition to the given file, VIL tries to load operating system specific files afterwards overriding previous settings, i.e., considering the given path as basis, augmenting it with an operating system specific infix (`win` for Windows, `macos` for MacOs and `unix` for all other Unix-like systems) before the filename extension. For example, in addition to `my.properties` on Windows, VIL also tries to load also `my.win.poperties` (no error will occur if operating specific properties are not given). Please note that the variables described in the properties file must be defined as global variables (outside any Rule) in the VIL script.
- Loading values from an external file ends by a semicolon.

Example:

```
load properties "globalVariables.properties";
Integer variableName;
```

Fills the global variables in the Script with values defined in the given properties file. Assuming that `variableName = 1` is stated in the properties file, the value of the variable `variableName` will be `1` after loading the properties.

3.1.8 Typedefs

Generic types such as the collections can lead to rather complex but inconvenient type names. A typical example is

```
mapOf(String, mapOf(String, String)) data;
```

In order to simplify the use of such types and to increase type safety, we allow the definition of type aliases akin to C or the unconstraint typedefs in IVML.

Syntax:

```
typedef name Type;
```

Description of Syntax: The declaration of typedefs consists of the following elements:

- The keyword **`typedef`** indicates the declaration of an alias type.
- The identifier *`name`* denotes the name of the new (alias) type.
- The *`Type`* is an existing type expression used to define the alias.

Type aliases can be used wherever the original type can be used.

Example:

```
typedef DataType mapOf(String, mapOf(String, String));
```

Then the expression shown above becomes

```
DataType data;
```

3.1.9 Rules

Build rules are used in VIL to specify individual production strategies, reusable build steps to be used within production strategies or, as the main entry point into the build process. Akin to *make* [8], VIL-rules may have preconditions, which must be fulfilled in order to enable the rule. However, VIL-rules may also explicitly define postconditions, which guard the result of the rule execution.

VIL rules may have **parameters** in order to parameterize the specified variability instantiation. These parameters must either be bound by the calling rule or, in case of the main entry rule, by the VIL script itself.

Preconditions may be given in terms of path-patterns, an individual artifact, an artifact collection or rule calls. While an arbitrary number of rule calls may be given as precondition, at most one path pattern, artifact or artifact collection may be given as first precondition[7]. If the preconditions of a rule are not fulfilled, the rule is not considered for evaluation, i.e., it also does not fail.

- A path pattern follows the (pattern) rules of ANT path specifications already described in Section 3.1.5.2. For example, "`$target/bin/**/*.class`" requires the existence of at least one Java bytecode file in the `bin` folder of `$target` (assuming that `$target` refers to the target project). Used as a rule precondition, a path pattern requires that the matching file artifacts exist and are up-to-date (akin to Make rule preconditions [8] but with extended pattern matching capabilities).

[7] Future work on VIL may relax this condition and even extend the current file path notation to more generic artifact model path expressions also involving fragment artefacts etc.

- An artifact (collection) is given in terms of a variable or a VIL expression evaluating to exactly one artifact (collection) instance. In a precondition, the denoted artifact(s) must exist and be up-to-date.
- A rule call (rule name with argument list) represents an explicit rule dependency and must be executed successfully in case that the preconditions of the stated rule are valid. The execution results of a rule call become available as an implicit variable in the rule body under the name of the called rule.
- A Boolean expression based on parameters, global variables or a (function) rule calls. If the @advice mapping (Section 3.3.11) is activated, also a VIL variable or a compound access evaluating to a Boolean value can be given.

The optional **rule postcondition** is given in terms of a path pattern, an individual artifact, an artifact collection or a Boolean expression. Postconditions are evaluated if the preconditions are met and the body of the rule is executed successfully. A rule completes successfully, if also the (optional) postcondition is met.

Rules may explicitly depend on each other in terms of the rule calls described above. Further, **implicit rule dependencies** are expressed via the first (non-rule call) pre- or postcondition (akin to *make* rules [8]). If a path matching precondition for rule r_0 is not fulfilled, the VIL execution environment will aim at fulfilling the precondition by (recursively) searching for rules r_i with a postcondition indicating that the successful execution rule r_i contributes to the unmet precondition of rule r_0. Ultimately, the possibly contributing rules r_i are executed (including their implicit rule dependencies) and the precondition of rule r_0 is checked again, and, on fulfilment, also r_0 is executed. If finally the precondition of r_0 is not fulfilled, r_0 is not considered for execution.

The rule body specifies the individual steps to be executed if the preconditions are met. A rule body may contain variable declarations, (assignment) expressions, explicit rule calls (not relevant as preconditions), instantiator calls, execution of system commands, or iterated execution of the previous elements. We will first describe the syntax of rules and describe then the individual statements available for specifying rule bodies.

If no path matching precondition is given, the rule body is executed once. If a path matching precondition is present, one or multiple artifacts may match that precondition and for each of these artifacts a corresponding output artifact may be required by the postcondition (if specified). However, the related match conditions may directly be used in the rule body but then possibly lead to a (superfluous) re-instantiation of the related target artifacts. In order to avoid re-instantiation and to allow for optimizing the variability instantiation, VIL offers two ways of executing the rule body. These two ways are:

1) Passing the right hand side matches as an implicit collection variable called `FROM_MATCH` to the rule body. This is more appropriate for instantiations which may operate on multiple artifacts and consider dependencies among artifacts by themselves, such as a Java compiler.
2) Implicitly iterating over the matched pairs of left hand and right hand side artifacts. Then rule body is executed iteratively over all matching

precondition artifacts. In order to address the actual artifact to be processed as well as its expected resulting artifact, the implicit variables[8] `TO` (in case of a matching precondition) and `FROM` (in case of a matching postcondition) will be made available to the loop body. This way is more appropriate for single artifact instantiations, such as calling a pre-processor or a C compiler.

By default, rules return implicitly their execution results consisting of two sequences,

- `result` containing the immediately modified artifacts by that particular rule. For a rule `r` you can retrieve the results explicitly by `setOf(Artifact) res = r().result();` or simpler `setOf(Artifact) res = r();`.
- `allResults` containing the modified artifacts by all dependent rule calls. For a rule `r` you can retrieve the results explicitly by `setOf(Artifact) res = r().allResults();`

In addition, rules may act as functions, i.e., a rule may declare a return type and the last expression executed by the rule determining the result value must comply with the return type.

Further, the artifacts modified by executed rules are be successively collected in the implicit global collection variable `OTHERPROJECTS`.

Syntax:

```
[protected] [Type] name (parameterList) = [[postcondition]
 : [preconditions]] {
 // TO/FROM/FROM_MATCH may be available in the body
 //variable declarations
 //rule, instantiator, artifact or system calls
 //iterated execution
}
```

Description of Syntax: A rule declaration consists of the following elements:

- The optional keyword **`protected`** prevents that this rule is visible from outside so that such rules cannot be used as an entry point (for example, in ANT [9] this is expressed by a target name starting with the minus character) or from other scripts.
- An optional type (see second syntax form above) may be given in order to declare a “function”. In this case, the last expression must return a value of the return type.
- The *`name`* allows identifying the rule for explicit rule calls or for script extension.
- The *`parameterList`* specifies explicit parameters which may be used as arguments for precondition rule calls as well as within the rule body.

[8] The old names RHS, LHS and RHSMATCH are still available, but deprecated. They will be removed in one of the next versions.

Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameters may have default values (*`Type name = value`*), but all default parameters must be given after parameters without default values. Parameter must either be bound by the calling rule or, in case of the main entry rule (usually called `main`), by the VIL script itself (same parameter sequence and assignable values in both, the template and the main sub-template).

- The first three parts may be omitted in case of anonymous rules which are only executed due to implicit dependencies and not available to explicit rule calls.
- The optional postcondition specifies the expected outcome of the rule execution. A postcondition may be a path match, an artifact or an artifact collection. In case of a path match, the implicit variable FROM will be made available to the rule body.
- The optional preconditions specify whether the rule is considered for execution. The first precondition (made available as implicit variable TO to the rule body) may be a path match, an artifact or an artifact collection. The following preconditions may be explicit rule calls. The execution results of the preconditions will be made available to the rule body in terms of implicit variables with names of the called rules and the rule return type described above. If neither pre- nor postcondition are given, also the separating colon can be omitted.
- The rule body is specified within the following curly brackets.

Example:

```
produceGenericCopy(FileArtifact x, FileArtifact y) = y : x
{
    x.copy(y);
}

compileGoal() = "$target/bin/*.class" : "$source/*.java"
{
    javac(FROM, TO);
}

Integer helper(Integer param) =
{
    param + 100;
}
```

The rule body specifies the individual steps to be performed in order to fulfil the rule postcondition (if stated). A rule body may contain variable declarations, (assignment) expressions, explicit rule calls, instantiator calls, execution of system commands or iterated execution of these elements. The statements (ended by a semicolon or a statement block) given in a rule body are executed in the given sequence. The

`helper` rule defines a “function” returning a derived value, in particular to enable the reuse of complex expressions. We will discuss these individual elements in the following subsections.

Rules can be annotated similar to Java, e.g.,

```
@dispatchBasis
Integer helper(Integer param) =
{
    param + 100;
}
```

We define the following annotations (case insensitive naming):

- `@override`: The annotated operation overrides an already existing operation with the same or a refined signature. An implementation shall emit at least a warning if there is no overridden base operation.
- `@dispatchBasis`: The annotated operation acts as basis for dynamic dispatch calls and, thus, typically utilizes the most generic types.
- `@dispatchCase`: The annotated operation overrides directly or transitively a dispatch basis operation. An implementation shall emit at least a warning if there is no overridden dispatch basis.

3.1.9.1 Variable Declarations

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within nested blocks. Basically, a variable declaration within a rule body follows the same syntax as global variable declarations discussed in Section 3.1.5.5.

3.1.9.2 Expressions

Expressions such as value calculations or execution of artifact operations may be used within a rule body as a guard expression or as a variable assignment. Please note that we will detail VIL expressions in Section 3.3, as the expression language is common to both, VIL and VTL.

- Guard expressions constrain the execution of the remaining statements in a rule body, i.e., the expression must be evaluated successfully in order to continue the execution of the rule.
- In a variable assignment, the expression on the right hand side of the assignment operator “=” must be evaluated successfully in order to assign the evaluation result of the right side to the variable specified on the left side.

As we will explain in Section 3.3 in more detail, expressions are evaluated lazily, i.e., expressions that are undefined, because a used value or operation call is evaluated to undefined are ignored and do not lead to the failing of containing rules.

3.1.9.3 Calls

A call leads to the execution of another rule, an instantiator or an artifact operation. We will discuss three types of calls in this section, as they are represented by the same syntax. However, the most extreme call of a (blackbox) instantiator, namely the execution of an operating system command (including operating system scripts) follows a slightly different syntax. We will discuss operating system commands in Section 3.1.9.4.

The syntax of rule calls, instantiators or artifact operations looks as follows:

```
operationName(argumentList)
```

whereby arguments are expressions separated by commas. Calls may return values of different type.

Rule Calls

An explicit rule call is stated in terms of the name of the rule and the arguments matching the parameter list of the target rule. A rule call leads to the execution of a rule defined in the same script, one of the extended scripts or an imported script. As rules with the same signature consisting of name and parameter list are shadowed by the extension, rules in extended scripts may explicitly be called by

```
super.operationName(argumentList)
```

Operations defined in imported scripts may be denoted by their qualified name as `operationName`, i.e., prepending the import path until the defining model.

Instantiator Calls

Basically, VIL aims at defining the production flow for instantiating generic artifacts for a software product line. In contrast, the VIL template language aims at specifying the individual actions to instantiate an individual (generic) artifact. Further instantiators may be given in terms of (wrapping) Java classes in order to make programming language compilers, linkers, or legacy instantiators available. Such instantiators may provide information about their execution, in particular the created artifacts.

In VIL, all these types of instantiators are mapped transparently to one kind of statement, the instantiator call:

```
operationName(argumentList)
```

Basically, an instantiator call looks similar to a rule call, i.e., a name with a parameter list, but it (typically) returns a collection of artifacts (or even nothing in case of wrapped blackbox instantiators). Instantiators may be rather generic (such as the built-in instantiator for the VIL template language) and may offer to pass an arbitrary number of arguments (e.g., those defined by a VIL template. Therefore, depending on the instantiator, named arguments (`parameterName = valueExpression`) may pass arbitrary VIL instances to an instantiator in a generic way.

We will detail the built-in instantiators in Section 3.4.9. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize an instantiator.

Artifact Operation Calls

Artifact operations provide information on an individual artifact, its fragments or even enable the manipulation of artifacts. Basically, an artifact operation is executed on a variable or expression, which evaluates to an artifact type. An artifact operation can be expressed (akin to IVML and OCL) in two different ways, using the artifact as first argument

```
operationName(artifact, argumentList)
```

or in object-oriented style

```
artifact.operationName(argumentList)
```

Basically, a String can be automatically converted into a Path or an `Artifact`. Similarly, a Path can be transparently converted into an `Artifact`. However, in some cases, also an explicit creation of an artifact of a certain type may be required. Typically, the individual artifact types support the following constructors

```
new ArtifactType(String)
```

for obtaining a specific artifact specified by its path. If the given *`ArtifactType`* does not support the actual format of the underlying artifact, e.g., a Java file shall be considered as an XML file, the resulting artifact is undefined and, due to lazy evaluation, subsequent expressions are ignored.

Artifacts are associated with creation rules. Basically, file artifacts (regardless of whether they physically exist or only the path is known) are polymorphically determined according to their file name extension, e.g., a file with extension `xml` is considered to be a `XmlFileArtifact`. However, pattern paths, i.e., paths containing wildcards, will not be turned into artifact instances. Further, content-specific rules may apply depending on the specific artifact type. If no such rule applies, a basic `FileArtifact` is created as the default fallback. Thus, the underlying mechanisms of the VIL artifact model will check whether the creation of that instance (regardless of whether the underlying file exists or not) is actually possible or not. If the creation fails, also the containing rule will fail.

Folders are created transparently, e.g., when the underlying file beyond a file artifact is created. The constructor

```
new ArtifactType()
```

allows to obtain a temporary file or folder artifact. Unless not renamed, this artifact will be automatically deleted after terminating the execution of the VIL script.

The modifications to a VIL artifact instance will automatically be synchronized with the underlying artifact upon the end of the lifetime of the related variable, e.g., when the execution of the containing scope of a local variable ends.

We will detail the built-in artifact operations in Section 3.4.8. Please refer to the developer documentation of EASy-Producer for obtaining information on how to realize own artifact types and related operations.

Operation Resolution

While determining the applicable rules, instantiators, or artifact operations, the VIL type system considers in the following sequence

(1) Exact match of argument types and parameter types.
(2) Assignment compatible argument types and parameters.
(3) Automatic conversions specified as part of the implementation of VIL types and artifacts, e.g., the implicit conversion of a `String` to a `Path` or a `String` to an `Artifact`. If multiple conversion operations may support the required conversion, the most specific one with respect to the type hierarchy of the source type will be applied. Details on the type system and the available conversions will be discussed in Section 3.3.

The operation types discussed in this section will be resolved according to the sequence below:

(1) Rule calls
(2) Instantiator calls
(3) Artifact, configuration type, and basic type operations

Further, the VIL runtime environment performs dynamic dispatch, i.e., the operation determined and bound at script parsing time will be reconsidered with respect to the actual types of parameters and the best matching operation will dynamically be determined (similar to dynamic dispatch in Xtend [2]). This avoids the need for explicit type checking or large alternative decision blocks that are painful to modify if the underlying type hierarchy changes. Moreover, as dynamic dispatch applies to importing and extended scripts, it supports openness of the instantiation, i.e., types that are added later, e.g., when configuring an IVML model, are considered and called dependent on the actual types and arguments.

To apply dynamic dispatch, a base case must be defined, i.e., a rule with the most common type to be used in that situation so that the language environment can initially bind the respective call statically. If further rules with the same signature but more specific types are defined, the dynamic dispatch mechanism will select the best match at execution time[9].

3.1.9.4 Operating System Commands

VIL is also able to execute the most basic form of a blackbox instantiator, namely operating systems calls or scripts. However, the syntax for system calls differs slightly from the other call types discussed in Section 3.1.9.3 as operating system commands may require explicit path specifications.

`execute` *`identifier`*`(`*`argumentList`*`)`

whereby *`identifier`* must denote a variable which evaluates to a `String` or a `Path`.

[9] As discussed in Section 3.1.4, VIL can utilize functions defined in VTL templates to facilitate reuse and to increase consistency. Currently, dynamic dispatch is not supported for a call from VIL to VTL. Thus, to exploit dynamic dispatch, a "frontend" VTL operation must be defined, which is called from VIL and performs the dynamic dispatch in VTL.

The *argumentList* contains the arguments to be passed to the external command in terms of a comma separated list. Although it is in VIL syntactically correct to put multiple arguments into a single String, this leads typically to execution problems as the underlying infrastructure (Java) expects individual arguments and even inserts quotes to cope with whitespaces.

This enables that operating system calls can be composed at script execution time or determined using external values. However, the related command or script is executed, but the created artifacts are not tracked by the VIL execution environment.

3.1.9.5 Alternative Execution

In VIL, alternatives allow choosing among different ways of instantiating artifacts, i.e., upon evaluating a condition the appropriate alternative to execute is determined.

The (return) type of an alternative is either the common type of all alternatives or `Any`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
if (expression) ifStatement
if (expression) ifStatement else elseStatement
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **`if`** indicates the beginning of an alternative statement.
- The *expression* given in parenthesis determines whether the if-part (condition is evaluated to true) or the else-part (condition is evaluated to false) is executed. If the *expression* cannot be evaluated, the alternatives will be evaluated.
- The *ifStatement* (or statement block enclosed in curly braces) is being executed when the *expression* is evaluated to true. A single statement must be terminated by a semicolon.
- The **`else`** part is optional, i.e., if else is used in an alternative, a following *elseStatement* is required. As usual, a dangling else is bound to the innermost alternative.
- The *elseStatement* (or statement block enclosed in curly braces) is executed if the *expression* is evaluated to false. Again, a single statement must be terminated by a semicolon.

Example:

```
if (config.variables().size() > 0) {
    // work on config
}
```

3.1.9.6 Iterated Execution

Finally, all statements available in a rule body may explicitly be executed in iterative fashion, e.g., to apply a sequence of instantiator calls explicitly to a collection of artifacts. This section introduces the VIL `map` expression, which is different from typical programming language loops as it collects (potentially in a parallel fashion) the result of its execution in terms of modified artifacts (similar to a rule). The `map` expression is more like a map-application in a functional language and, therefore, an implementation shall issue a warning if the loop accesses variables defined outside the loop, i.e., the body depends on other values and is potentially not executable as a single unit / not parallelizable. For typical loops see the for-loop in Section 3.1.9.7 and the while-loop in Section 3.1.9.8. The results produced by a map expression are determined by the last standalone expression in a map body. The evaluation results of these expressions are collected in a sequence of type of the expression. The results can be assigned to a variable. If there is no such expression, map will behave like a typical for-loop. However, due to its character as an expression, a map must be used within an expression and terminated by a semicolon.

Syntax:

```
map(names = expression) {
//variable declarations
//rule, instantiator, artifact or system calls
//iterated execution
};
```

Description of Syntax: An iterated execution consists of the following elements:

- The keyword **`map`** followed by parenthesis defining the iterator variables.
- The *`names`* denoting the names of the variables used by the map expression iterate. The number, type and the contents of the iterator variables are implicitly defined by the related `expression`. Optionally, in order to ease the learning for programmers, for each variable a type can be given which must comply with the respective inferred type from `expression`. Typically, the expression will lead to a collection with one parameter type so that map will iterate over that collection using exactly one variable of the element type of the collection. However, as we will discuss in Section 3.1.9.7, the `join` expression may return a multi-dimensional collection, which then needs multiple iterator variables.
- The equality/assignment symbol separates the variables from the `expression`. In order to ease the application and the learning for programmers, alternatively a colon may be used here.
- The **`map`** consists of a block determining the statements to be executed in for an individual iteration. The *`names`* denoting the iterator variables shall be used within the block.

Example:

```
map(d = config.variables()) {
    // operate on the iterator variable d of type
    // DecisionVariable (see Section 3.4.7.6)
};
```

Below we depict the equivalent notation using a colon, an explicit type and the implicit conversion of a decision variable to a collection using its contained values. Please note that explicit types for the variables and the use of the colon are independent from each other.

```
map(DecisionVariable d : config) {
    // operate on the iterator variable d of type
    // DecisionVariable (see Section 3.4.7.6)
};
```

3.1.9.7 For Loop

In addition to the `map` expression, VIL also offers a typical for loop based on iterating over a collection. A for loop can be terminated by a semicolon. A for loop does not return a result.

Syntax:

```
for(names = expression) {
//variable declarations
//rule, instantiator, artifact or system calls
//iterated execution
};
```

Description of Syntax: An iterated execution consists of the following elements:

- The keyword **`for`** followed by parenthesis defining the iterator variables.
- The *`names`* denoting the names of the variables used by the map expression iterate akin to a **`map`** expression. The number, type and the contents of the iterator variables are implicitly defined by the related `expression`. Optionally, for each variable a type can be given which must comply with the respective inferred type from `expression`.
- The equality/assignment symbol separates the variables from the `expression`.
- The **`for`** loop consists of a block determining the statements to be executed in for an individual iteration. The *`names`* denoting the iterator variables shall be used within the block.

Example:

```
for(d = config.variables()) {
    // operate on the iterator variable d of type
```

```
    // DecisionVariable (see Section 3.4.7.6)
};
```

Below we depict the equivalent notation using a colon, an explicit type and the implicit conversion of a decision variable to a collection using its contained values. Please note that explicit types for the variables and the use of the colon are independent from each other.

```
for(DecisionVariable d : config) {
    // operate on the iterator variable d of type
    // DecisionVariable (see Section 3.4.7.6)
};
```

3.1.9.8 While-Loop

The while-statement in VIL enables the defined repetition of statements based on a condition. Basically, it is rather similar to an iterator-loop in Java.

Syntax:

`while (`*`expression`***`)`** *`statement`*

Description of Syntax: A while-loop statement consists of the following elements:

- The keyword **`while`** indicates the beginning of a while-loop statement. It is followed by a parenthesis defining the loop condition, i.e., a Boolean expression indicating whether the loop body shall be executed.
- The statement (or statement block enclosed in curly braces) represents the loop body.

Example:

```
Integer i = 0;
while (i < 10) {
    //operate on i
}
```

3.1.9.9 Join Expression

One specific expression in VIL is particularly intended to be used with the `map` iteration statement, namely the `join` operation. However, as `join` is an expression, it may be used as an usual expression, e.g., on the right hand side of a value assignment to a variable.

This operation is inspired by database joins, e.g., as usual in SQL. The `join` operation allows combining collections of different VIL types, in particular elements from the variability configuration with source or target artifacts. Depending on type of the specified expression types, the `join` operation returns a typed sequence containing the results.

Syntax:

`join(`*$name_1$`:expression`*$_1$**`, `***$name_2$`:expression`*$_2$**`) with (`***`expression`***`)`**

Description of Syntax: A VIL join expression consists of the following elements:

- The keyword **`join`** followed by one parenthesis defines the collections to be joined and the related iterator variables (*name₁, name₂*).
- *name₁, name₂* denote variables used by the join expression iterate over the collections given in *expression₁* and *expression₂*. Without further restriction, the result will be a collection of pairs on the types parameterized by the types of *expression₁* and *expression₂*. The keyword `exclude` used before one of the names leads to a left- or right-sided join, thus restricting number of parameters of the resulting collection to one.
- The third `expression` specifies the join condition, i.e., an expression involving *name₁* and *name₂* to select the relevant results from the cross product of *name₁* and *name₂*, and to effectively reduce the size of the result.

Example:

```
// work on those decisions and artifacts where a certain
// string composed from the decision name occurs in the
// artifact (and may be substituted by an instantiator)
map(d, a :
  join(d:config.variables(), a:"$source/src/**/*.java")
  with (a.text().matches("${" + d.name() + "}")) {
    // operate on decision variable d and
    // related artifact a
}
```

3.1.9.10 Instantiate Expression

In addition to the instantiation of individual projects, VIL is specifically intended to support the instantiation of hierarchical and multi product lines. Therefore, it is necessary to execute scripts in other projects, if required on projections of the configuration and with specific target project. Basically, applying already known VIL concepts, it is possible to explicitly call the main rule of predecessor projects statically, i.e., to import the predecessor projects and to call the respective main rule through its qualified name. However, the references to the predecessor projects would be static and not subject to possible variability. Thus, we introduce the instantiate expression, which allows executing a VIL rule in a project allowing to dynamically referring to VIL scripts in (other) projects. Per se, the instantiate expression does not imply a recursion over predecessor projects, but if the predecessor project realizes recursive instantiation, the instantiate expression will perform the recursion. Please note, that due to the dynamic resolution, rule calls via the instantiate expression imply a higher overhead.

Syntax:

`instantiate` *name* **`(`***argumentList***`)`** *[***`with`** `(`*expression*`)]*

Description of Syntax: A VIL instantiate expression consists of the following elements:

- The keyword **`instantiate`** followed by either the name of a variable containing the project to instantiate or a string containing the qualified name of the rule to be executed.
- The argument list of the rule to be executed in parenthesis. Please note, that in particular also the source project (in form 1-3 the project the script is defined for), the (relevant part of the) configuration and the target project (typically the calling project) must be given (following the VIL parameter conventions). Further, optional named arguments may be given.
- Finally, the version specification of the model to be instantiated may be given.

Example:

```
// instantiate the predecessor projects into target
vilScript a (Project source, Configuration config,
  Project target) {
  map(Project p: source.predecessors()) {
    instantiate p (p, config, target);
  };
}
```

3.2 VIL Artifact/Template Language

In this section, we describe the concepts and language elements of the VIL template language in detail. In contrast to VIL, which aims at specifying the instantiation of all artifacts of a product line, the VIL template language aims at specifying the instantiation or generation of a single artifact.

3.2.1 Reserved Keywords

In the VIL template language, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by those of the common VIL expression language given in Section 3.3.1.

- **`@advice`**
- **`@format`**
- **`@indent`**
- **`<CR>`**
- **`const`**
- **`def`**
- **`default`**
- **`else`**
- **`extends`**
- **`extension`**
- **`for`**
- **`flush`**
- **`if`**
- **`import`**
- **`insert`**
- **`protected`**
- **`switch`**
- **`typedef`**
- **`template`**
- **`version`**
- **`while`**
- **`with`**

3.2.2 Template

The template (**`template`**) is the top-level structure in the VIL template language. This element is mandatory as it defines the frame for specifying how to instantiate a certain artifact. Please note that exactly one template must be given in a VIL template file.

The definition of a template requires a name, which acts for referring among VIL templates and a parameter list specifying the expected information from the calling VIL script such as the actual configuration and the target artifact (fragment). Please note that these two arguments must be provided to all VIL template scripts.

Basically, VTL may refer to all visible configuration settings in a variability configuration, more precisely to those actual values of decision variables (and their underlying structure), which are frozen. In order to make this integration explicit,

these decision variables may be directly referenced in the VIL template language by their qualified IVML name. As IVML configurations may be partial or even dynamically composed, the actual structure of a variability model is not necessarily known at the point in time when the VIL script is specified. Thus, the validity of qualified IVML identifiers can only be determined at execution time of the VIL script when also the actual configuration is known. To support the domain engineer in specifying valid templates, also the VIL template language provides the **`advice`** annotation (see also Section 3.1.2).

Optionally, a VTL template may extend another VTL template, i.e., reusing and extending production strategies by overriding (akin to object-oriented languages).

VTL particularly aims at supporting generative and manipulative instantiation of generic artifacts. Therefore, the VTL provides capabilities for easily specifying and generating contents. However, as usual in software development, also VIL templates shall be formatted properly. In order to distinguish between intended formatting and whitespaces that shall not occur in the target artifact (fragment), the VIL template language is able to take the actual indentation into account (as specified in the **`indent`** annotation). Taking the formatting of the templates into account avoids postprocessing of the results, e.g., by formatting mechanisms [2]. We will discuss the indentation processing of the VIL template language as part of the content statements in Section 3.2.9.7.

Akin to programming languages, VIL templates may contain (global) variable declarations as well as sub-templates (similar to methods in object-oriented programming languages or functions in the structured programming paradigm).

Syntax:

```
//imports
//functional extensions
@advice(ivmlProjectName)
@indent(indentationSpec)
@format(formatSpec)
template name (parameterList) [extends name1] {
  //optional version specification
  //variable definitions, type defs, compound defs
  //sub-template declarations
}
```

Description of syntax: The definition of a VIL template consists of the following elements:

- Optionally, imported templates or functional extensions by Java classes are listed first.
- Optional advices declaring the underlying variability models. This annotation is similar to VIL (see Section 3.1.2).

- An optional indentation annotation enabling the VIL template execution to take the actual indentation into account when processing content statements. We will detail the use of the indentation annotation in Section 3.2.9.7 along with the content statement, which actually considers indentation information.
- An optional formatting specification annotation, in particular to define the line ending of the target artifact. We will detail also the use of the format annotation in Section 3.2.9.7 along with the content statement, which actually considers indentation information.
- The keyword **`template`** defines that the following identifier *`name`* defines a new artifact instantiation template.
- The parameter list denotes the arguments a VIL template requires when being executed. Basically, a VIL-template receives the underlying variability configuration and the target artifact as parameters. If given as first parameters in this sequence, the parameters will be bound independently of their name and, thus, the parameters can be named arbitrarily. Otherwise (due to the option of additional named parameters as mentioned below), the sequence of the parameters may be arbitrary but the parameters must exactly be named “config” and “source”. The arguments are subject to dynamic dispatch, i.e., either the most generic type `Artifact` may be used for the target artifact or a more specific type can be used. In the latter case, the instantiator statement in the VIL script must also pass in a type-compliant artifact instance. Additional parameters may be defined which then must be stated in the calling VIL script as named arguments. Parameters may have default values given in syntax *`Type name = value`* listed after all parameters not declaring default values. An implementation shall treat unspecified parameters as undefined, i.e., expressions using this parameter are undefined.
- A VIL template may optionally extend an existing (imported) VIL template. This is expressed by **`extends`** $name_1$, where *`name1`* denotes the name of the extending script.
- The optional version specification, variable declarations and sub-templates are then stated within the curly brackets.

Example:

```
@advice(YMS)
template DbInit (Configuration config, Artifact target){
  /* Go on with description of the artifact instantiation
  starting with a main sub-template and possibly further
  (imported) sub-templates */
}
```

3.2.3 Version

Akin to IMVL and VIL, also the VIL template language can be tagged with an explicit version number in order to support evolution. The syntax for the version declaration is identical to VIL as discussed in Section 3.1.3.

3.2.4 Imports

The description of the instantiation of a certain artifact type may be defined in a single VIL template (possibly including sub-templates) or may be composed from reusable sub-templates specified in other (existing) scripts. Therefore, VIL templates may be imported. As VIL, also VTL supports wildcard imports and the experimental `insert` imports. In order to support also the evolution of product line build specifications, also the VIL template language allows the specification of version-restricted imports. Imports make the sub-templates defined in the specified build file accessible to the importing template. The syntax of imports in VIL templates is identical to imports in the VIL scripts as discussed in Section 3.1.4. Akin to VIL scripts, cyclic imports are not allowed and shall lead to error messages.

3.2.5 Typedefs

Akin to VIL, also in the VIL template language typedefs can be defined to simplify the use of complex types. Syntax and semantics for typdefs is identical to VIL as discussed in Section 3.1.5.5.

3.2.6 Functional Extension

Sometimes, it is necessary to realize specific supporting functions such as calculations in terms of a programming language rather than in the template language itself. Therefore, similar to Xtend [2], the VIL template language enables external functions in terms of static Java methods, to call these methods from the template language and to use the results in VIL templates. Basically, the realizing classes are declared in VIL as extension and containing static methods are made available as they would be VIL operations[10]. However, methods with already known signatures will not be redefined.

Syntax:

```
extension name;
```

Description of Syntax: A functional extension in the VIL template language consists of the following elements:

- The keyword **`extension`** followed by a qualified Java *`name`* denoting the class to be considered. The referred class must be available to VIL through class loading. Contained static methods will be considered as extension methods for the VIL template language. Please note that the implementing

[10] Additional classes may require special treatment in terms of class loading, in particular in OSGi and Eclipse due to specific class loaders per bundle. In such environments, the class loader being responsible for the extension classes must be explicitly registered with the VIL runtime environment (see `de.uni_hildesheim.sse.easy_producer.instantiator.model.templateModel.ExtensionClassLoaders`).

method shall use only primitive Java types or (the implementation classes of the) VIL types discussed in Section 3.3.

- An extension declaration ends with a semicolon.

Example:

```
extension java.lang.System;
```

The statement above makes all static Methods of the class `java.lang.System` available to VTL.

3.2.7 Types

Basically, the VIL template language is a statically typed language with some convenience in terms of postponed type checking at runtime akin to VIL. Thus, the VIL template language provides a set of formal types available for variable declarations or parameter lists. VIL template language and VIL rely on the same type system and, thus, the VIL template language provides the same types as discussed in Section 3.1.5.

3.2.8 Variables

A variable provides named access to a value of a certain type similar to variables in programming languages. The semantic of variables as well as the syntax for declaring and using them in the VIL template language is identical to VIL as discussed in Section 3.1.5.5 (except for the capability of defining variable values in an external file which is not available in the VIL template language).

Similar to VIL, variables may be referred in Strings such as paths or content statements. A variable reference looks like `$variableName`. Entire VIL expressions (see Section 3.3) including variables may be given in the form `${expression}` (may be escaped by prefixing a backslash, i.e. \\ prevents the respective replacement and causes that `$variableName` or `${expression}` is emitted as given without the escaping backslash). When applying the respective element, variable and expression references are substituted by their actual value.

3.2.9 Sub-Templates (defs)

The actual instantiation of an artifact is given in terms of sub-templates (called `def` in the concrete syntax), i.e., named functional units with parameters and return types. One specific sub-template (usually called `main`) acts as the entry point into artifact instantation. Akin to VIL, it receives the parameters of the containing template as arguments (in the same sequence).

The body of a sub-template specifies the individual steps to be executed for realizing the instantiation. Such a body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). We will first describe the syntax of templates and discuss then the statements available in sub-template bodies.

Syntax:

```
[protected] def [Type] name (parameterList) {
//variable declarations
//alternative, switch-case, loop
//content statements
}
```

Description of Syntax: A sub-template declaration consists of the following elements:

- The optional keyword **`protected`** prevents that this rule is visible from outside so that such rules cannot be used as an entry point or from other scripts.
- The keyword **`def`** indicates the definition of a sub-template.
- By default, the VIL template language aims at inferring the return type of a sub-template from the rule body. In the extreme case, individual statements may produce a rather generic value of type `Any`, which enables the use of the value without type checking at template parsing time and type checking at runtime. However, in some situations the template developer may explicitly want to do strict type checking at template parsing time. This is enabled by specifying the optional return type for a sub-template.
- The *`name`* allows identifying the sub-template for calls, template extension or as `main` entry point.
- The *`parameterList`* specifies the parameters of a sub-template in order to parameterize the instantiation operations subsumed by the respective sub-template. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameters may have default values (*`Type name = value`*), but all default parameters must be given after parameters without default values. Parameters must either be bound by the calling sub-template or, in case of the main entry rule, by the VIL template itself (via identical names and assignable types, the template and the main sub-template).
- The rule body is specified within the following curly brackets.

Example:

```
def main(Configuration config, Artifact target) {
    // define artifact instantiation
}

def String valueMapping (DecisionVariable var) {
    // map the value of var to a String
    // explicit type checking is enforced
}
```

The sub-template body specifies the individual steps needed to instantiate an artifact. Such a rule body may contain variable declarations, (assignment) expressions, alternatives, switch-case-statements, loops and content statements (for producing the actual content). The statements (ended by a semicolon or a statement block) given in a sub-template body are executed in the given sequence. We will discuss these individual statement types in the following subsections.

The last statement executed in a sub-template body implicitly determines the return value of a sub-template. Please note that returning a String requires an expression such as a switch (Section 3.2.9.4) or a variable declaration, as otherwise a String is recognized as a Content statement (Section 3.2.9.7). Please note that a variable reference / expression `${expression}` can call a def.

Defs can be annotated similar to Java, e.g.,

```
@dispatchBasis
Integer helper(Integer param) =
{
    param + 100;
}
```

We define the following annotations (case insensitive naming):

- `@override`: The annotated operation overrides an already existing operation with the same or a refined signature. An implementation shall emit at least a warning if there is no overridden base operation.
- `@dispatchBasis`: The annotated operation acts as basis for dynamic dispatch calls and, thus, typically utilizes the most generic types.
- `@dispatchCase`: The annotated operation overrides directly or transitively a dispatch basis operation. An implementation shall emit at least a warning if there is no overridden dispatch basis.

3.2.9.1 Variable Declaration

A variable declaration within a rule body introduces a local variable shadowing rule parameters or global variables. This is in particular true for variables, which are defined within statement-blocks such as loops or alternatives. Basically, a variable declaration within a sub-template body follows the same syntax as global variable declarations discussed in Section 3.2.8.

3.2.9.2 Expression Statement

Expressions such as value calculations or execution of artifact operations may be used within a sub-template body as a guard expression or as a variable assignment. Please note that we will detail the VIL expression language in Section 3.3, as the expression language is common to both, the VIL instantiation language and the VIL template language. Thus, guard expressions and variable assignments as discussed in Section 3.1.9.2 are similarly available in the VIL template language. Further, similar call types as well as their (resolution) semantic as discussed in Section 3.1.9.3 are

available in the VIL template language (of course, rule calls are replaced by template calls and operating system calls by calls to functional extensions). Template calls may be recursive. In the VIL template language, the resolution sequence is

1) Template calls
2) Artifact, configuration type and basic type operations
3) Functional extension calls

3.2.9.3 Alternative

In the VIL template language, alternatives allow choosing among different ways of instantiating an artifact, i.e., upon evaluating a condition the appropriate alternative to execute is determined.

The (return) type of an alternative is either the common type of all alternatives or `Any`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
if (expression) ifStatement
if (expression) ifStatement else elseStatement
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **`if`** indicates the beginning of an alternative statement.
- The *`expression`* given in parenthesis determines whether the if-part (condition is evaluated to true) or the else-part (condition is evaluated to false) is executed. If *`expression`* cannot be evaluated, the alternatives will be evaluated.
- The *`ifStatement`* (or statement block enclosed in curly braces) is being executed when the *`expression`* is evaluated to true. A single statement must be terminated by a semicolon.
- The **`else`** part is optional, i.e., if else is used in an alternative, a following *`elseStatement`* is required. As usual, a dangling else is bound to the innermost alternative.
- The *`elseStatement`* (or statement block enclosed in curly braces) is executed if the *`expression`* is evaluated to false. Again, a single statement must be terminated by a semicolon.

Example:

```
if (config.variables().size() > 0) {
    // work on config
} else {
    // produce an empty artifact
}
```

3.2.9.4 Switch

The switch expression in the VIL template language is for (dynamically) mapping configuration elements to artifact elements rather than for influencing the control flow (as it is the case for the alternative statement). However, in case of larger mappings with (more or less) static content, we suggest using a map variable (see Section 3.1.5.4).

The (return) type of an alternative is either the common type of all cases or `Any`. The return value of an alternative is determined by the last statement executed in the alternative selected by the (condition) expression.

Syntax:

```
switch (expression) {
  expression1 : expression2,
  expression3 : expression4,
  [default : expression5]
}
```

Description of Syntax: An alternative statement consists of the following elements:

- The keyword **`switch`** indicates the beginning of a switch expression. It is followed by an *`expression`* to switch over and the individual cases in a block of curly brackets.
- A case consists of an expression (*`expression`*$_1$ or *`expression`*$_3$ above) to be matched against expression. If an individual match succeeds, the related value expression will be evaluated and determines the (result) value of the switch expression. The implicit variable `VALUE` may be used within the value expression in order to refer to the evaluated value of *`expression`*.
- Optionally, a **`default`** case can be given which is considered if none of the previous cases matches. In that case, the expression stated behind the default will be evaluated and determines the (result) value of the switch expression.

Example:

```
switch (var.name()) {
    ”forkNumber” : VALUE + var.intValue() – 1,
    ”cpuNumber” : var.stringValue()
    //go on with further cases and a default value
    //if required
}
```

3.2.9.5 For-Loop

The for-statement in VTL enables the defined repetition of statements based on iterating over a collection.

Syntax:

for (Type *var* **:** *expression***)** *statement*

for (Type *var* **:** *expression*, *expression*$_1$**)** *statement*

for (Type *var* **:** *expression*, *expression*$_1$, *expression*$_2$**)** *statement*

Description of Syntax: A for-loop statement consists of the following elements:

- The keyword **for** indicates the beginning of a for-loop statement. It is followed by a parenthesis defining the loop iterator, i.e., a variable to which successively all values of the *expression* are assigned. Therefore, *expression* must either evaluate to a set or a sequence.
- The statement (or statement block enclosed in curly braces) is then executed for each element in the collection specified by *expression* while the iterator *var* is successively assigned to each individual value in the collection.
- An optional separator expression *expression*$_1$ which is emitted (without line end) at the end of each iteration if further iterations will happen. Such a separator expression simplifies generating value lists or similar target artifact concepts.
- A second optional separator expression *expression*$_2$ which may only be stated if *expression*$_1$ is given. *expression*$_2$ is emitted (without line end) at the end of the last iteration, but not if no iteration happened. Such a separator expression simplifies generating value lists or similar target artifact concepts.

Example:

```
for (DecisionVariable var: config.variables()) {
    //operate on var
}
```

Akin to VIL, VTL supports the implicit conversion of a decision variable to its contained variables, i.e., we may omit the `variables()` operation.

```
for (DecisionVariable var: config, ", ") {
    '${var.name()}'
    // (the separator will be added if needed)
}

for (DecisionVariable var: config, ",", ";") {
    '${var.name()}'
    //(the separator will be added if needed,
    //the second separator will be printed after the final
    //iteration)
}
```

3.2.9.6 While-Loop

The while-statement in VTL enables the defined repetition of statements based on a condition. Basically, it is rather similar to an iterator-loop in Java.

Syntax:

```
while (expression) statement
```

Description of Syntax: A while-loop statement consists of the following elements:

- The keyword **`while`** indicates the beginning of a while-loop statement. It is followed by a parenthesis defining the loop condition, i.e., a Boolean expression indicating whether the loop body shall be executed.
- The statement (or statement block enclosed in curly braces) represents the loop body.

Example:

```
Integer i = 0;
while (i < 10) {
    //operate on i
}
```

3.2.9.7 Content

The content statement is used to generate the content of the target artifact. Basically, all characters given in a String (enclosed in a pair of apostrophes or quotes including appropriate Java escapes and line breaks) are emitted as output to the result artifact. Content statements executed in the course of template evaluation according to the control flow make up the entire content of the target artifact (fragment). A content statement may consist of multiple lines as part of the content. Thereby, variable references or IVML expressions are substituted as described for variables in Section 3.2.8.

Chained (sequential) content statements within the same def are concatenated. By default, content statements within a def are ended by a line break, except for the last one, to which a line end is appended by default only in the top-level def (can be adjusted using explicit line ends such as `\n` or the `<CR>` keyword in conjunction with the content statement as introduced below in this section). For returning the concatenated result from a def, the last statement of the respective def must be the last content statement of the sequence (also the last statement in the last alternative/loop counts). Multiple def calls can be combined within a content statement stating the defs as substitutable expressions. Please note that the return of all defs involved in such chains must return/end with a respective content statement so that all results can be properly collected in the sequence of execution. Moreover, it is important to note that in a chain of def calls content statements are only chained, if every def ends with a content statement. In particular, if one def delegates the work, e.g., as it consists only of a loop and a (dynamic dispatched) def call, the respective call must be given as a substitutable expression within a content statement, e.g., `'${furtherDev(someElement)}'`.

Without further consideration, also the indentation whitespaces for pretty-printing a VIL-template will be taken over into the resulting artifact. In order to provide more control about the formatting, the annotation **`@indent`** allows specifying the number of whitespaces used as one indentation step (value `indentation`), the number of whitespaces to be considered in tabulator emulation (value `tabEmulation`) and also how many additional whitespaces (value `additional`, default is `1`) are used to indent the content statement, i.e., whether the following lines after the lead in character are further subject to indentation or not. Further whitespaces in the content are considered as formatting of the content itself and are taken over into the artifact. In addition, an optional numerical value can be specified at each content statement in order to programmatically indent the configuration by the given number of whitespaces. If you rely on automatic formatting and call other VTL templates, those templates shall also specify an indentation hint as otherwise the indentation in the called template are not considered correctly / ignored.

In particular, if you rely on the automatic handling of VTL indentations activated by **`@indent`**, in some situations, a differently formatted output would be desirable, but, which cannot be derived from the indentation structure of your VTL code. One particular situation is that a substitutable expression evaluates to an empty string, which is appended to the actual indentation, leading to an empty line that may not be desirable (or which is even not permissible in the target artifact). While the formatting is correct from the viewpoint of VTL, it is not desirable. To handle such situations, we introduced **formatting hints**, a brief optional addition at the end of a substitutable expression (within the curly brackets), which tells VTL what specific formatting to apply. Currently, VTL supports two formatting hints:

1) `${…|e}` if the expression evaluates to an **e**mpty string, emit an **e**mpty line, i.e., clear the preceding indentation and immediate line break(s). This formatting hint is not applied if non-whitespace text was emitted before the expression.
2) `${…|<}` remove the preceding indentation (not applied if non-whitespace text was emitted before the expression). Usually, the same effect can be achieved by chaining content statements, i.e., denoting the respective expression in an own content statement indicating that the result shall be emitted at the beginning of a line.

In some cases, it is required to explicitly define the line ending of the target artifact, e.g., when operating system scripts shall be generated or manipulated. Therefore, the annotation **`@format`** allows specifying the formatting. Akin to **`@indent`**, the formatting is defined as key value pairs. Currently, VTL supports exactly the key **`lineEnd`**, which receives a String value defining the line end, e.g., **`@format(lineEnd = "\r\n")`** for a forced Linux line end. Formatting options are

- **`lineEnd`**: The explicit string used for line ends. If not stated, the system default determined by Java is used.
- **`lineLength`**: Optional maximum line length activating the VTL contents formatter to ensure the line length. By default, indentation is not considered

when splitting lines. Specific profiles may enable this capability. The value can be given as String or as number.

- **`charset`**: The desired output character set. Must be a valid Java charset name, e.g., UTF-8. If not given, the system's default character set is used.
- **`profile`**: A pre-defined formatting profile. If not given, a simple default profile for general text is applied. Currently, only the "Java" profile is defined, which enables taking the indentation configuration into consideration to indent split lines based on the previous indentation. Single line comments and Javadoc comments are specifically considered. Further, multiple empty lines are removed when this profile is activated.
- **`profileArg_`*`name`***: Determines the string/int value of a profile-specific argument. For the "Java" profile, the argument "javadocIndent" defines the lead in to be inserted when splitting JavaDoc comments into multiple lines, by default " * ".

Syntax:

```
"text" [[[!] <CR>] [| expression];]

'text ' [[[!] <CR>] [| expression];]
```

- **Description of Syntax**: A content statement consists of the following elements:The lead in / lead out marker (apostrophe or quote) indicates the content statement and marks the actual content. Two forms are used to enable the use of the opposite character in content. A content statement may cover multiple lines of content and always ends with a line break.
- By default, line ends are emitted at the end of a content statement only if used in top-level defs, while defs called from other defs are considered as reusable units and are not ended by a line end. The optional `<CR>` indicates[11] that the content shall end with a line break, while !`<CR>` after a content indicates that there shall not be a (default) line break. The no-linebreak form of the content statement is also helpful in combination with the separator expression in loops or if a String (without linebreak ending) shall be returned from a VTL def. If `<CR>` is given, the content statement must end with a semicolon.
- An optional indentation expression can be indicated by the pipe symbol and followed by a numerical *`expression`*. The numerical *`expression`* determines the amount of whitespaces to be used as mandatory indentation prefix for each individual line of the actual content statement. Indentation is only considered if the expression evaluates to a positive

[11] Until version 0.98 this was stated using the keyword `print`. However, stating that no line break shall occur before a content statement is not intuitive, disturbs the code layout for explicit formatting and may prevent further global commands such as `println`.

number, i.e., 0 or a negative number may be given, but this is then ignored. Only if the indentation expression (may be a constant or a true expression) is specified, a semicolon is required.

Example:

```
'CREATE DATABASE ${var.name()}'
'CREATE TABLE data' | 4;
'INSERT INTO data' !<CR> | 4;
```

While the first content statement emits the text with expression substitution as it is, the second content statement indents the text to be emitted by 4 whitespaces. The third statement also indents the content by 4 whitespaces, but no explicit line end is emitted (please note that emitting no line end is the default behaviour for nested def calls).

Within a content statement, VTL expression evaluation also supports in-content alternative and in-content loop, i.e., expressions that represent content selection and content iteration. Although both in-content statements can also be expressed using VTL alternatives (Section 3.2.9.3) or VTL loops (Section 3.2.9.5), in several situations it is convenient to use the respective in-content statement (as also usual in other template languages). This is in particular helpful in combination with explicit formatting, where using VTL alternatives or VTL loops may require to explicitly inserting line breaks to have full control over indentation. The syntax for the two in-content statements is:

- Alternative:
  '${IF** *condition***}***then-part***${ELSE}*else-part***${ENDIF}'**
  Alternatives can be one-sided (without else-part) or two-sided. The *condition* must evaluate to a Boolean expression. The markers indicating the alternative are always stated in capital letters and in variable reference / expression syntax. If the syntax is not complete or erroneous, just the content of *then-part* and *else-part* is emitted as hint.
- Loop:
  '${FOR** *var* **:** *init* **SEPARATOR** *ex* ***END*** *ex***}***body-part***${ENDFOR}'
  The loop initialization expression *init* must evaluate to a collection. The element type of the initialization collection is taken as type for the loop iterator variable *var*, i.e., no explicit typing is allowed here. As for VTL loops (Section 3.2.9.5), the colon (**:**) separating iterator variable and initialization expression can be alternatively be stated as an equals character (=). The *body-part* is evaluated for each element in *init*. The result of the loop expression is the concatenation of the results of the evaluations of the body-part for all elements in *init*. The **SEPARATOR** expression, which must evaluate to a String, is optional, i.e., keyword and expression can be omitted, and defines the characters used to separate the evaluations of the *body-part* for individual elements of *init*. If the **SEPARATOR** expression is given, also an optional **END** expression can be given. The end separator (must also evaluate to a String) is emitted after the last iteration, but only if there was at least

one iteration. The markers indicating the alternative are always stated in capital letters and in variable reference / expression syntax. If the syntax is not complete or erroneous, then nothing is emitted as *body-part* may contain references to the then unbound iterator variable.

In-content alternatives and in-content loops can be nested, although it is recommended replace complex combinations by nested defs. If an in-content expression is finally evaluated to an empty string and indentation support is enabled via **`@indent`**, the preceding indentation is removed, i.e., it appears in case of an empty alternative / loop result as if that expression has not been specified.

3.2.9.8 Content Flush

The results of content statements as introduced in Section 3.2.9.7 are collected and written into the script parameter artifact by VTL once at the end of a script execution. If, however, during a script execution the result produced so far shall be persisted, e.g., to explicitly execute the complete formatting of an artifact (if supported by the respective artifact type), then the content must be written explicitly. This can be achieved by the `flush` statement. Alternatively, formatting of the respective artifact can be triggered by the controlling VIL script.

Syntax:

```
flush;
```

Description of Syntax: A content flush is stated by the keyword flush and a trailing semicolon.

Example:

```
flush;
```

3.2.9.9 Builder Block Expression

When creating contents through specialized artifacts, often long changes of function calls occur, e.g., for the Java code artefact

```
c.method("int", "calculate").return("0");
```

While such functions shall be defined in a way that chaining is possible, it does not always work, e.g., if method parameters can be configured further such as adding an annotation. Then, a variable, in the example above for the method, must be declared, requiring detailed knowledge on respective types. However, often it would be sufficient to declare a local variable, in the case above of the method type, within a block allowing further statements to operate with that variable, while at the end of that block returning the value of that variable to allow for further chaining. This is the particular aim of the builder block expression, a combination of a nested block and an expression, which can only be used in call chains (earliest as second chain element), passing through the value of call before, making the value locally available within its block.

Syntax:

```
ex.(({statements-using-o}))[.call]*

ex.(v|{statements-using-v})[.call]*

ex.(Type t|{statements-using-t})[.call]*
```

In the syntax summary above, the builder block is the second element of the call chain, while there must be at least one variable use or call before (denoted here as `ex`) and potentially arbitrary calls after. The block of this expression may be empty.

- The first form takes the value of `ex` calculated before the builder block expression and passes it into the block, declared as implicit local variable named `o` of the type of the value. The statements within the block may use o to access the value of `ex` (as well as any other reachable variable). Ultimately, the builder block expression returns the value of `o` for further chaining. Existing variables named `o` are shadowed.
- The second form explicitly names the implicit variable, here as `v`. Statements within the block may use `v` to access the value of `ex`. As above, the value of the builder block expression is the value in `v` for further chaining. Typically, this form is used most frequently.
- The third form explicitly names and types the implicit variable, here as `t`. The Statements within the block may use `t` to access the value of `ex`. As above, the value of the builder block expression is the value in `v` for further chaining. The value of the builder block expression is undefined if the value of `ex` cannot be assigned to `t`.

Example:

```
c.method("int", "calculate").(m|{
      m.param("int", "a");
      m.param("int", "b");
}).return("a + b");
```

In the example (based on the Java code artefact), the declaration of the method parameters `a` and `b` happens within the builder block, which defines an implicit variable of method type named "m". The return expression of the method is chained after the block. Chaining the return specification after the block is just for illustration purposes. Typically, one would define the return within the block based on `m`. Chaining is more adequate, e.g., to chain a method call after a method call declaring a (variant) argument list.

3.3 VIL Expression Language

In this section, we will define the syntax and the semantics of the VIL expression language, which is common to the VIL instantiation language and the VIL template language. Expressions in the VIL languages are inspired by IVML, i.e., similar concepts and operations shall be available to state expressions in VIL if IVML is already known (and vice versa). The VIL expression language is mostly taken over from IVML, i.e., a systematic selection of OCL driven by the concepts and types of IVML and VIL. Moreover, VIL defines additional operations that ease artifact modification and code generation, and is therefore inspired by QVT [15]. Subsequently, most of the content in this section is taken from OCL [6] or the IVML language specification [3, 7] and adjusted to the need, the notational conventions, and the semantics of the VIL languages.

3.3.1 Reserved Keywords

Keywords in the VIL expression language are reserved words. That means that the keywords must not occur anywhere in an expression as the name of a rule, a template or a variable. The list of keywords is shown below:

- **`abstract`**
- **`and`**
- **`callOf`**
- **`compound`**
- **`false`**
- **`new`**
- **`not`**
- **`or`**
- **`refines`**
- **`sequenceOf`**
- **`setOf`**
- **`super`**
- **`true`**
- **`xor`**
- **`null`**

In order to increase reuse among the VIL languages, the VIL expression language also provides the definition of common language concepts such as variable declarations and parameter lists. The related keywords were already listed in Section 3.1.1 and 3.2.1, respectively.

3.3.2 Prefix operators

The VIL expression language defines two prefix operators, the unary

- Boolean negation '**`not`**' and its alias '!'.
- Numerical negation '−' which changes the sign of a Real or an Integer.

Operators may be applied to constants, (qualified) variables or return values of calls.

3.3.3 Infix operators

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

```
a + b
```

is conceptually equal to the expression:

```
a.+(b)
```

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

Operators may be applied to constants, (qualified) variables or return values of calls. Two relational comparison combined by a Boolean, i.e., an expression of form *a rOp1 b and b rOp2 c* and can be equally expressed by the shorted range comparison *rOp1 b rOp2 c*, e.g., `a <= b` **`and`** `b <= c` as `a <= b <= c`.

Please note that, while using infix operators, in VIL Integer is a subclass of Real. Thus, for each parameter of type Real, you can use Integer as the actual parameter. However, the return type will always be Real. We will detail the operations on basic types in Section 3.4.3.

3.3.4 Precedence rules

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

- dot operations: '**.**' for operation calls and compound slot / IVML enum literal access in object-oriented style. Alternatively, as in OCL, enum literals can also be qualified by '**.**'. The latter may lead to an OCL compliance warning in a concrete implementation.
- unary '**not**', !(alias for **not**) and unary minus '-'
- '*' and '/'
- '+' and binary '-'
- '<', '>', '<=', '>='
- '==' (equality), '<>', '!=' (alias for '<>')
- '**and**', '**or**' and '**xor**'

'(' and ')' can be used to change precedence.

3.3.5 Datatypes

All artifacts defined by the extensible VIL artefact model as well as the various built-in types are available to the expression language and may be used in expressions. IVML elements are mapped into VIL via IVML qualified names. Figure 1 illustrates the

VIL type hierarchy (not detailing the IVML integration through the configuration types). Below, we discuss the use of datatypes.

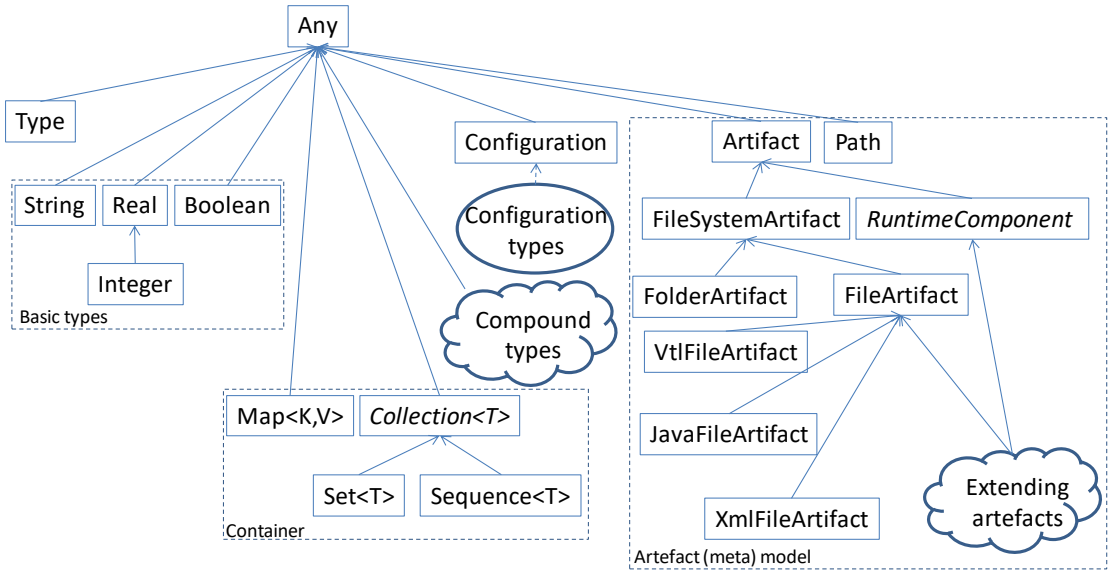

Figure 1: Overview of the VIL type system

3.3.6 Type conformance

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

- `Any` is the common superclass of all types. All types comply with `Any`. However, `Any` is not directly available to the user and used internally to denote expressions of unknown type to be resolved and checked while execution time of the specifying script.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Collections are parameterized types regarding the contained element type. Collections comply only if they are of the same collection type and the type of the contained elements complies (checked before script evaluation time) or if no type parameters are given (deferred to script evaluation time).

3.3.7 Side effects

In contrast to OCL, some constraint expressions in IVML may lead to side effects, in particular to modifications of artifacts and artifact fragments.

3.3.8 Undefined values

Basically, variables and IVML qualified names, i.e., links into a variability model, may be undefined. During evaluation of expressions at script or template execution time,

undefined (sub-)expressions are ignored and do not lead to failing rules or sub-templates. This is in particular true for VTL content statements (see Section 3.2.9.7), which are not emitted if any subexpression is undefined.

3.3.9 Null

An IVML variable may have the explicit value `null`, i.e., it is configured with `null`. In VIL / VTL, the keyword `null` represents the `null` value in IVML. This can be helpful to figure out, whether a variable was configured with null as shown below (for VTL):

```
if (var == null) {
    // ...
}
```

From the perspective of the type system[12], `null` is a specific value of `Any`, i.e., it is not a pointer to nowhere as in many programming languages. Thus, `null` is defined and does not lead to undefined values and ignored expressions (see Section 3.3.8). Please note that a decision variable configured with `null` is not undefined from the point of view of IVML as it has a value.

3.3.10 Collection operations

The VIL artifact model and the IVML integration into VIL in terms of the `Configuration` type define many operations returning collections. Operating with collections is specifically meant to enable a flexible and powerful way of accessing information and for deriving artifacts.

Practically, collections in VIL are sets, sequences, or maps as introduced in Section 3.1.5.4. Collections are a basis for iterations in the VIL template language (Section 3.2.9.5) as well as for joins and map iterations in VIL (Section 3.1.9.6). Therefore, we defined a set of basic operations on the artifact types and the `Configuration` providing convenient access through selection and filtering. This is expressed in terms of iterator expressions, which are called through the "->" operator rather than the dot operator "." As in OCL, all VIL collection operations can be called through the "->" operator and, as in IVML, alternatively through the "." operator (an OCL compliance mode realized by an implementation may warn about the latter case). Collection operations require a collection of values to operate on, i.e., either the expression the operation is called for is a container or the value/variable is implicitly turned into a Set of the type of the value/variable.

One example is the **select** operation, which returns all elements of a collection complying to a certain Boolean selection expression. An example for an expression with an iterator variable[13] is

```
configuration.variables()->select(i|i.name().length()== 10)
```

[12] Please note that the notation `(null == var)` is (currently) not considered as a valid operation as `Any` does not define operations.

[13] The old notation using the generic iterator name `ITER` still works for compatibility. The new notation is akin to IVML, but the explicit type of an iterator variable is stated in VIL/VTL before the name of the variable. The syntax allows for multiple iterators, but currently only one iterator variable is supported.

or with implicit (inferred) iterator variable for exactly one unbound variable due to the collection type

```
configuration.variables()->select(i.name().length()== 10)
```

or with explicit types (of course the type must match the collection type)

```
configuration.variables()->select(
  DecisionVariable i|i.name().length()== 10)
```

Both expressions return those decision variables with a name of length 10. In addition to select, also a generic sort operation is provided for sequences.

Other examples for collection iterator expressions are the **select**, **reject**, **collect** and **iterate**[14] operations also known from IVML / OCL. While select and reject return sub-collections, collect returns the value of an expression for each element of a collection, and iterate allows defining arbitrary iterative expressions that may lead to a primitive or collection result. Following the example shown above, the *basic form* of iterate is

```
configuration.variables()->iterate(Integer r = 0; i|
  r = r + i.name().length())
```

The first declarator defines the aggregator, the second the iterator over the collection, while the expression modifies the aggregator successively. In this form, the result type of iterate corresponds to the type of the aggregator and its value is the value of the aggregator at the end of the iteration. Please note that aggregators must be initialized with a default value, while iterators must not have a default value (implicitly taken from the collection an iterator is applied to). The short form of iterate does not define an aggregator and, thus, just iterates over the collection without result to be returned. Assuming that `files` is a collection of `FileArtifact`, then the following expression deletes all those artifacts.

```
files->iterate(f|f.delete())
```

Some operations can provide implicit collections so that iterator variables and dependent expressions can be applied similarly. One example is the deleteStatement operation provided by the JavaMethod discussed in the VIL extensions document. Given such a Java method, one can delete all JavaCalls matching the condition given by the iterator expression as shown below:

```
method->deleteStatement(JavaCall c |
  c.type() == "EASyLogger" and c.name() == "exception");
```

3.3.11 Dynamic extension of the type system through IVML

Basically, VIL / VTL are loosely integrated with the variability model, i.e., generic VIL / VTL types allow accessing IVML variables and configurations, e.g., via the name of a variable. The generic access is beneficial, if the variability model itself may change dynamically (a potential future capability) or to iterate over the structure of the variability model and derive artifact based on the structure. However, in standard

[14] The apply operation still exists, but an alias for iterate was introduced with the OCL alignment in version 0.98.

situations having a fixed variability model, using the generic VIL / VTL types may be inconvenient.

Until version 0.9 of VIL / VTL, the `@advice` annotation enabled to use IVML identifiers directly in VIL / VTL instead of strings for variable names or enumeration values. This supports the specification of VIL / VTL scripts as external names are checked by the editor. Since version 0.92, the `@advice` annotation makes also IVML types available to VIL / VTL. As in IVML, qualified names composed by `'::'` can be used to refer to IVML types, slots and enum literals (alternatively enum literals can be accessed by `'.'`, which may lead to an OCL compliance warning in a concrete implementation). For example, let

```
project Demo {
  compound MyCompound {
    Integer slotA;
  }
  MyCompound myCompound;
}
```

be an IVML variability model. Accessing the variable `slotA` in `myCompound` looks in VIL/VTL using the generic types as follows

```
DecisionVariable comp = config.byName("myCompound");
Integer i = v.byName("slotA");
```

or after using `@advice(Demo)` also

```
DecisionVariable comp = config.byName(myCompound);
Integer i = v.byName(MyCompound::slotA);
```

Using `@advice(Demo)` in VIL/VTL version version 0.95 [15]

```
Demo d = config;
Integer i = d.myCompound.slotA;
```

Similarly, also the project annotations can be accessed. Moreover, and this was the initial motivation for extending the `@advice` annotation, IVML types can directly be used in rule and template signatures in order to exploit dynamic dispatch (see Section 3.1.9.3) to process refined IVML compounds in an extensible manner. As a result, differentiating types can easily be expressed in terms of rules or templates using the same signature but specialized parameter types for all involved IVML types.

In more detail, the following rules apply if an `@advice` annotation is used:

- IVML types are mapped transparently to VIL / VTL types. IVML types from projects specified in `@advice` annotations are mapped with their simple type name (if the name has not been defined before in VIL) and with their qualified name. In particular, contained variables are represented as fields of the specific IVML type (in contrast to the generic integration via

[15] To improve and clarify the mapping, the old mapping implemented in versions 0.92 to 0.94 has been disabled.

`DecisionVariable)`. Please note that the value of fields cannot be changed in VIL.

- IVML projects are mapped transparently to VIL / VTL types to provide seamless access to the top-level IVML variables. A VIL / VTL configuration instance can directly be assigned to a VIL / VTL variable representing an IVML project type (checked at runtime) and the representing IVML type also to the VIL / VTL configuration type. A VIL/VTL variable representing an IVML project type can be assigned back to a variable of `Configuration` type. Thereby, the configuration is projected to the respective (sub-)project.
- Annotations of an IVML project are mapped as fields of the specific types defined by the IVML project. Annotations defined on individual variables are mapped to the type. However, inconsistent individual definitions will be mapped only once (for the first type in sequence). In case that individual annotations with different types are used, the generic VIL / VTL types shall be used to access these annotations. Please note that the value of annotations cannot be changed in VIL.
- All types which are dynamically introduced to VIL / VTL via `@advice` annotations define the (generic) operations from `DecisionVariable` (see Section 3.4.7.3 for details).

Due to implicit conversions, one can deliberately switch between the specific IVML types and the generic types. However, it is advisable to follow one style in order to simplify the VIL/VTL specifications. Akin to the lightweight checking of identifiers introduced by `@advice`, i.e., unknown identifiers are reported by the editors as a warning, also unknown types and operations on unknown types are not checked strictly and indicated by a warning. However, unknown types are considered as undefined expressions (see Section 3.3.8) and, thus, not evaluated.

3.3.12 Function Types

In some cases, passing a function (VIL rule or VTL def) into a function as parameter is helpful, e.g., to make execution parts reusable while still being parameterizable. This is in particular useful if script / template inheritance and overriding the respective functions is not intended or possible. VIL offers for this purpose a specific type indicating a function signature (akin a function prototype in C). Basically the syntax for a function without return type is `callOf(`*`Type1`*`, `*`Type2`*`, ...)` with *`Type1`*, *`Type2`* the types of the formal parameters and `callOf` *`RType`* `(`*`Type1`*`, `*`Type2`*`, ...)` with *`RType`* the return type. Function types can be used as parameters or in typedefs, i.e., for defining an alias type name for a function type. A valid value for a function type parameter is a function matching the specified types.

Example (VTL, akin for VIL):

```
typedef FunCall callOf Integer (Integer);
FunCall fun = func;
Integer func(Integer i) = : {
    i * 10;
}
```

```
Integer call(FunCall f, Integer i) = : {
    f(i) + 1;
}
Integer use(Integer i) = : {
   call(fun, i) + call(func, i);
}
```

The example above illustrates the use of function types. The typedef defines an alias type with name `FunCall` representing a function receiving an integer parameter and returning an integer value. The function `func` is an example for such a function (many more with different name may exist) and, thus, is a valid value for the variable `fun` of the function type `FunCall`. The function `call` is a function, which is parameterized by a function of type `FunCall` in terms of the parameter `f` without detailing what `f` will actually calculate. As we know the signature of `f` through the type `FunCall`, we can just use it, here in the expression `f(i) + 1`. If `call` is called, a function realizing `f` will be passed in as parameter and can be called / evaluated by the expression. To demonstrate the application, function `use` calls the function `call` in two different ways, first via the function stored as value in the variable `fun`, second by directly passing the previously defined function `func` as parameter (in this example, in both cases, the same function will be called on the same input and in both cases the same result will be evaluated). Wherever we use in the xample the alias type `FunCall`, we could also directly use the function type **`callOf`** `Integer (Integer)`, but defining and using an alias type is more convenient and less error prone. As alias types and visible functions defined by imported scripts are available in the importing script, the implementing functions used to call such parameterized functions can also be defined in different scripts.

3.4 Built-in operations

Similar to OCL and IVML, all operations in VIL are defined on individual types and can be accessed using the "." operator, such as `set.size()`. However, this is also true for the equality, relational, and mathematical operators but they are typically given in alternative infix notation, i.e., 1 + 1 instead of 1.+(1) as stated in Section 3.3.3. Further, the unary negation is typically stated as prefix operator. Due to the VIL artifact model, the integration with IVML and the specific purpose of variability instantiation, the VIL types define a different set of operations than OCL/IVML.

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

Please note that those operations starting with "get" (Java-getters) are also available with their short name without "get" in order to simplify script and template creation, e.g., the `getName()` operation is also available as its aliased operation `name()`. We will make this explicit by listing both operation signatures.

3.4.1 Global operations

Some operations are of global character and can be considered independent of a certain type. These operations are

- **enableTracing(Boolean b)**
  Enables or disables tracing from the current position to the next call of this operation or to the end of the script depending on b.
- **notifyProgress(Integer a, Integer m, [String d])**
  Notifies about the actual instantiation progress. a represents the actual step number, m the maximum step number (may vary at runtime), d is an optional description. The background is that due to loops and recursions we cannot determine automatically the maximum number of operations or whether the script will terminate at all. Whether a progress indication is displayed depends on the integration, e.g., headless integrations may ignore this information.
- **println(Any a)**
  Prints the String representation of a to the

3.4.2 Internal Types

This section summarizes the internal VIL types. Alias operations, i.e., operations with same return type, parameter types but different name are listed (without return type) after a slash after the primary operation.

3.4.2.1 Any

`Any` is the most common type in the VIL type system. All types in VIL are type compliant to `Any`. In particular, `Any` is used as type if the actual type is unknown at parsing time and shall be determined dynamically at runtime. Therefore, `Any` can be assigned to variables of any type (but no specific operations can be executed on `Any`). `Any` can be converted automatically into a `String`, `Real` `Integer` or `Boolean`.

- **Boolean == (Any r) / != (Any a)**
  True if the *operand* is the same as *r*.
- **Boolean <> (Any a)**
  True if the *operand* is different from *a*.
- **String getLocale (Any a) / locale (Any a)**
  Returns the (global) locale via *operand* as "*language*" or "*language_country*". As in IVML / OCL, the default locale is "en_US", but may be redefined by the implementing tool, e.g., via properties.
- **Type getType()**
  Returns the type of *operand*.
- **Boolean isTypeOf (Type t)**
  True if the *operand* has the same type as *t*.
- **Boolean isKindOf (Type t)**
  True if the *operand* has the same type as *t* or is a subtype of *t*.
- **String locale(String s)**

Changes the (global) locale via *operand* and returns the actual locale. S shall be given as "*language*" or "*language_country*".

3.4.2.2 Type

The type Type represents type expressions themselves and enables the type-generic selection type-compliant elements from collections. Types can be given as qualified or simple names as required, e.g., through the dynamic integration of IVML models via @advice.

- **Boolean == (Type t)**
  True if the *operand* is the same as *t*.
- **Boolean <> (Type t) / != (Type t)**
  True if the *operand* is different from *t*.
- **setOf(?) allInstances ()**
  Returns all instances of the *operand* in the top-level configuration passed into VIL/VTL. If no instances are known, an empty set is returned (always for basic types). The actual type of the returned set complies to the underlying type, in particular IVML compound types mapped via @advice into IVML.
- **String toString ()**
  Returns the type name of the *operand*.
- **String name() / getName ()**
  Returns the type name of the *operand*.
- **String qualifiedName() / getQualifiedName ()**
  Returns the qualified type name of the *operand*.

A type can be converted automatically into its name.

3.4.2.3 Void

The Void type represents expressions leading to no type.

3.4.3 Basic Types

In this section, we detail the operations for the basic VIL types.

3.4.3.1 Real

The basic type `Real` represents the mathematical concept of real numbers following the Java range restrictions for double values. Note that `Integer` can automatically be converted to `Real`.

- **Boolean == (Real r)**
  True if the *operand* is the same as *r*.
- **Boolean <> (Real r) / != (Real r)**
  True if the *operand* is not the same as *r*.
- **Boolean < (Real r)**
  True if the *operand* is less than *r*.
- **Boolean > (Real r)**
  True if the *operand* is greater than *r*.

- **Boolean <= (Real r)**
  True if the *operand* is less than or equal to *r*.
- **Boolean >= (Real r)**
  True if the *operand* is the same as *r*.
- **Real + (Real r)**
  The value of the addition of *self* and the *operand*.
- **Real - (Real r)**
  The value of the subtraction of *r* from the *operand*.
- **Real - ()**
  The negative value of the *operand*.
- **Real * (Real r)**
  The value of the multiplication of the *operand* and *r*.
- **Real / (Real r)**
  The value of the *operand* divided by *r*. Leads to an evaluation error if *r* is equal to zero.
- **Real abs()**
  The absolute value of the *operand*.
- **Integer ceil()**
  The closest integer value that is greater or equal to the *operand*.
- **Integer floor ()**
  The largest integer that is less than or equal to the *operand*.
- **Boolean isFinite()**
  Returns whether the *operand* is a finite real.
- **Boolean isInfinite()**
  Returns whether the *operand* is an infinite real.
- **Boolean isNaN()**
  Returns whether the *operand* is not a number.
- **Real max(Real r)**
  The maximum value of the *operand* and *r*.
- **Real min(Real r)**
  The minimum value of the *operand* and *r*.
- **Real random()**
  Returns a random number 0 and 1.0 (exclusive). No operand is needed.
- **Integer round()**
  The integer that is closest to *the operand*. When there are two such integers, the largest one.
- **String toString ()**
  The string representation of *operand*.

3.4.3.2 Integer

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

- **Boolean == (Integer i)**
  True if the *operand* is the same as *i*.

- **Boolean <> (Integer i) / != (Integer i)**
  True if the *operand* is not the same as *i*.
- **Boolean < (Integer i)**
  True if the *operand* is less than *i*.
- **Boolean > (Integer i)**
  True if the *operand* is greater than *i*.
- **Boolean <= (Integer i)**
  True if the *operand* is less than or equal to *i*.
- **Boolean >= (Integer i)**
  True if the *operand* is greater than or equal to *i*.
- **Integer + (Integer i)**
  The value of the addition of the *operand* and *i*.
- **Integer - (Integer i)**
  The value of the subtraction of *i* from the *operand*.
- **Integer - ()**
  The negative value of the *operand*.
- **Integer * (Integer i)**
  The value of the multiplication of the *operand* and *i*.
- **Real / (Integer i)**
  The value of the *operand* divided by *i* including remainder. Leads to an evaluation error if *i* is equal to zero.
- **Integer abs()**
  The absolute value of the *operand*.
- **Integer div (Integer i)**
  The value of the *operand* divided by *i* without remainder. Leads to an evaluation error if *i* is equal to zero.
- **Integer max (Integer i)**
  The maximum value of the *operand* and *i*.
- **Integer min (Integer i)**
  The minimum value of the *operand* and *i*.
- **Integer modulo (Integer i)**
  The modulo value of the *operand* and *i*.
- **Integer randomInteger()**
  Returns a random integer between 0 and 1.0 (exclusive). No operand is needed.
- **Integer randomInteger(Integer m)**
  Returns a random integer between 0 and m-1 (exclusive) using m as operand (to distinguish from `randomInteger()`.
- **String toString ()**
  The string representation of *operand*.

Conversions: `Integer` values can automatically be converted to `Real` values.

3.4.3.3 Boolean

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

- **Boolean == (Boolean a)**

True if the *operand* is the same as *a*.

- **Boolean <> (Boolean a) / != (Boolean a)**
  True if the *operand* is not the same as *a*.
- **Boolean ! ()**
  True if *self* is false and vice versa. Alias for !()
- **Boolean and (Boolean b)**
  True if both *b1* and *b* are true.
- **Boolean iff (Boolean b)**
  Shortcut for (*operand*.implies(*b*) and *b*.implies(*operand*)).
- **Boolean implies (Boolean b)**
  True if *operand* is false, or if *operand* is true and b is true. Please note that in contrast to IVML, VIL realizes the plain Boolean implication.
- **Boolean not ()**
  True if *self* is false and vice versa.
- **Boolean or (Boolean b)**
  True if either *self* or *b* is true.
- **String toString (Boolean b)**
  The string representation of *operand*.
- **Boolean xor (Boolean b)**
  True if either *self* or *b* is true, but not both.

3.4.3.4 String

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

- **Boolean == (String s)**
  True if the *operand* is the same as *s*.
- **Boolean <> (String s) / != (String s)**
  True if the *operand* is not the same as *s*.
- **String + (Any s)**
  The concatenation of the *operand* and (the Java String representation of) *s*. In particular, this operation concatenates two Strings.
- **String + (Path p)**
  The concatenation of the *operand* and the string representation of path *p*.
- **Boolean < (String s)**
  True if the *operand* is less than *s* considering the current locale.
- **Boolean <= (String s)**
  True if the *operand* is less than or equal *s* considering the current locale.
- **Boolean > (String s)**
  True if the *operand* is greater than *s* considering the current locale.
- **Boolean >= (String s)**
  True if the *operand* is greater than or equal *s* considering the current locale.
- **sequenceOf(String) characters()**
  Returns the characters of *operand* as sequence of strings.
- **String concat (Any s)**
  The concatenation of the *operand* and (the Java String representation of) *s*. In particular, this operation concatenates two Strings.

- **Boolean endsWith(String s)**
  Returns whether *operand* ends with suffix *s.*
- **Boolean equalsIgnoreCase (String s)**
  True if the *operand* is the same as *s* ignoring upper/lower characters considering the current locale.
- **String firstToLowerCase() / firstToLower()**
  Turns the first character of *operand* into a lower case character considering the current locale.
- **String firstToUpperCase() / firstToUpper()**
  Turns the first character of *operand* into an upper case character considering the current locale.
- **String format(Any v)**
  Formats the format string in *operand* using the value(s) in *v*. Basically, the format string uses the Java format operation, i.e., the related string format specification. As in QVT, *v* can be a map (then %(key) is replaced by the corresponding value in the map), a sequence for passing multiple values to a format string formatting multiple values or (everything else is interpreted as) a single value. If the format specification does not match the given value(s), the result is undefined (and ignored).
- **Integer indexOf(String s) / find(String s)**
  Returns the leftmost position of *s* within operand, -1 if *s* is not substring of *operand*.
- **Boolean isQuoted(String q)**
  Returns whether operand has form *q* + *o* + *q* and, therefore, is quoted by *q*.
- **Integer length () / Integer size ()**
  The number of characters in the *operand*.
- **String replace(String s, String r)**
  Returns *operand* with all substrings *s* replaced by *r*.
- **Integer rfind(String s)**
  Returns the rightmost position of *s* within operand, -1 if *s* is not substring of *operand*.
- **sequenceOf(String) split (String r)**
  Splits the *operand* along the regular expression r. Regular expressions are given in the Java regular expression notation. If r does not match an empty collection is returned.
- **Boolean startsWith(String p)**
  Returns whether *operand* starts with prefix *p.*
- **String substitute(String r, String p)**
  Returns *operand* with all substrings matching the Java regular expression *r* substituted by *p*.
- **String substring(Integer s)**
  Returns the substring of the *operand* from *s* (inclusive) to the end of *operand*. *operand* is returned in case of any problem, e.g., *s* exceeding the valid index range.
- **String substring(Integer s, Integer e)**

Returns the substring of the *operand* from *s* (inclusive) to *e* (exclusive). *operand* is returned in case of any problem, e.g., positions exceeding the valid index range.

- **String substringBefore(String m)**
  Returns the substring in *operand* before the first match of *m*. Returns an empty string in case of no match.
- **String substringAfter(String m)**
  Returns the substring in *operand* after the first match of *m*. Returns an empty string in case of no match.
- **Boolean matches (String r)**
  Returns whether the *operand* matches the regular expression *r*. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether `mail` is a valid e-mail-address:
  ```
  mail.matches([\w]*@[\w]*.[\w]*);
  ```
- **String normalizeSpace()**
  Returns a copy of *operand* with all trailing/leading spaces removed and internal multi-spaces replaced by a single space each.
- **sequenceOf(String) tokenize (String d)**
  Tokenizes the *operand* along the delimiters d. This is a simplified form of the `split` operation.
- **String lastToLowerCase() / lastToLower()**
  Turns the last character of *operand* into a lower case character considering the current locale.
- **String lastToUpperCase() / lastToUpper()**
  Turns the last character of *operand* into an upper case character considering the current locale.
- **Boolean matchBoolean ()**
  Whether *operand* represents / can be converted into a Boolean value.
- **Boolean matchReal ()**
  Whether *operand* represents / can be converted into a Real value.
- **Boolean matchIdentifier ()**
  Whether *operand* represents / can be converted into an identifier using `toIdentifier()` without removing characters.
- **Boolean matchReal ()**
  Whether *operand* represents / can be converted into an Integer value.
- **Boolean toBoolean ()**
  Converts the *operand* to a Boolean value (turning all strings not indicating true to false).
- **Integer toInteger ()**
  Converts the *operand* to an Integer value.
- **String toIdentifier()**
  Turns the operand into a typical (Java) programming language identifier, e.g., by removing illegal characters such as spaces. Cases of the individual characters will not be changed. The result may be empty.
- **String toLowerCase() / toLower()**

Turns the *operand* into a `String` consisting of lower case characters considering the current locale.

- **Real toReal ()**
  Converts the *operand* to a Real value.
- **String toUpperCase() / toUpper()**
  Turns the operand into a `String` consisting of upper case characters considering the current locale.
- **String trim()**
  Returns a copy of *operand* with all trailing/leading spaces removed.
- **String getProperty (String k, String d)**
  Returns the value of the Java property *k*, *d* if the key is not defined/unknown.
- **String osName()**
  Returns the (Java) name of the underlying operating system.
- **String content(String s)**
  A VTL template returning an empty content statement may be represented by a non-printable string marker. This method returns either *s* or, if *s* is the marker, an empty string.
- **String quotify(String q)**
  Quotifies *operand*, i.e., returns *q* + *operand* + *q*.
- **String unquotify(String q)**
  Quotifies *operand*, if operand has form *q* + *o* + *q* this operation returns *o* else *operand*.
- **String stripEnd(String s)**
  Removes all characters in *s* in any sequence from the end of *operand*.
- **String stripStart(String s)**
  Removes all characters in *s* in any sequence from the start of *operand*.

3.4.4 Compound Types

Compound types are defined through VIL statements as introduced in Section 3.1.5.5. All slots defined by a compound are accessible through the dot operation. In addition, a compound defines the following operations.

- **String getName() / name()**
  Returns the type name of *operand*.
- **String toString()**
  Returns the type name of *operand*.
- **Type getType() / type ()**
  Returns the type of *operand*.

Compounds can be converted automatically to the name of the compound using the toString operation. If not abstract, a compound instance can be created through the compound constructor, named parameters can be used to assign specific values to the compound slots.

3.4.5 Collection Types

This section defines the operations of the collection types. VIL defines one abstract collection type `Collection` and two specific collections, namely `Set` and

`Sequence`. All collection types are parameterized by one parameter. Below, 'T' will denote the parameter for the collection types. A concrete collection type is created by substituting a type for the parameter T. So a collection of integers is referred in VIL by `setOf(Integer)`. Although the keyword `collectionOf` does not exist, we will use it in this section to denote types of Collection (Section 3.4.5.1)

In addition, VIL defines the type `Map`, an associative container, which allows relating keys to values.

3.4.5.1 Collection

`Collection` is the abstract super type of all collections in VIL.

- **Boolean ==(collectionOf(T) c) / equals(collectionOf(T) c)**
  Returns whether *operand* contains the same elements than *c*, for ordered collections such as `Sequence` also whether the elements are given in the same sequence. The other operations defined by Any are available, too.
- **T add (T e)**
  Adds the element e to the set *operand*.
- **T any (Expression e)**
  Returns an element in *operand* that complies with the iterator *e*.
- **Any avg ()**
  Returns the sum of all elements of the operand if the element type supports the binary +, / or div operations. The result is undefined if no such operation is defined on the element type. The actual result depends on the collection type, in case of numeric element types the result is typically Real.
- **clear()**
  Removes all entries from *operand*.
- **collectionOf(T) clone()**
  Returns a cloned shallow copy of *operand*.
- **setOf(T) closure (Iterator | expression)**
  Returns the transitive closure of the elements (of reference type) specified by the expression. As in OCL, it always returns a (flat) set of results. See also `isAcyclic`.
- **collectionOf(A) collect (Expression e)**
  Returns the flattened[16] results of applying the iterator expression *e* to all elements in *operand*. The result is flattened.
- **containerOf(T) collectNested (Iterator | expression)**
  The container of elements that results from applying *expression* to every member of the *source* set. Nested collections remain in the result.
- **forEach (Expression e)**
  Applies the iterator expression *e* to the flattened[16] collection of *operand*. No result is returned.
- **Integer count (T object)**
  The number of times that *object* occurs in the collection *operand*.

[16] The explicit flattening was introduced in version 0.98 along with the OCL alignment. The original operation did not consider creating nested structures, leading to the same result in most cases.

- **Boolean excludes (T o)**
  True if *o* is not an element of *operand*, false otherwise.
- **Boolean excludesAll (collectionOf(?) c)**
  True if none of the elements in *c* are also in *operand*, false otherwise.
- **Boolean exists (Expression e)**
  Returns whether an element in *operand* exists that complies with iterator *e*.
- **collectionOf(?) flatten()**
  Turns *operand* into a flat collection. In case that *operand* is already flat and does not contain any sub-collections, this operation does not have any effect. Otherwise the flattened collection is returned, i.e., sub-collections are resolved to their containing plain elements.
- **Boolean forAll (Expression e)**
  Returns whether all elements in *operand* comply with the iterator *e*.
- **Boolean includes (T o)**
  True if *o* is an element of *operand*, false otherwise.
- **Boolean includesAll (collectionOf(?) c)**
  True if all elements from *c* are also in *operand*, false otherwise.
- **setOf(T) isAcyclic (Iterator | expression)**
  Returns whether the transitive closure of the elements (of reference type) specified by the expression does not contain a cycle. See also `closure`.
- **Boolean isUnique (Expression e)**
  Returns whether all results returned by *e* are unique.
- **Boolean isEmpty ()**
  Is the *operand* the empty collection?
- **Boolean isNotEmpty () / notEmpty ()**
  Is the *operand* not the empty collection?
- **? iterate (Expression e) / apply(Expression e)**
  Applies *e* to all elements in *operand*. Without aggregating declarator, nothing is returned and just a loop over all elements is executed. If an aggregator is declared, the operation returns a result of the type of the aggregator (denoted as *A* above).
- **Boolean remove (T e)**
  Removes e from *operand* and returns whether *e* was removed.
- **T product ()**
  Returns the sum of all elements of the operand if the element type supports the binary * operation. The result is undefined if no * operation is defined on the element type.
- **collectionOf(T) reject (Expression e)**
  Returns the elements in *operand*, which do not comply with the iterator *e*.
- **Integer size ()**
  The number of elements in the collection *operand.*
- **collectionOf(T) select (Expression e)**
  Returns the elements in *operand*, which comply with the iterator *e*.
- **collectionOf(T) sortedBy(Expression e)**

Sorts the elements in *operand* according to the respective values defined by the iterator expression *e*. In case of numerical values, numerical order is considered, else lexicographical order using the default locale.

- **T sum ()**
  Returns the sum of all elements of the operand if the element type supports the binary + operation. The result is undefined if no + operation is defined on the element type.
- **setOf(T) toSet() / asSet()**[17]
  Turns *operand* into a set (excluding duplicates).
- **sequenceOf(T) toSequence() / asSequence()**
  Turns *operand* into a sequence.
- **T one (Expression e)**
  Returns the only element in *operand* that complies with the iterator *e*. The result is undefined if there are multiple matches through *e*.

No automated conversions are defined among collections of generic configuration types (e.g., `DecisionVariable`, cf. Section 3.4.7.3) and specific configuration types mapped into VIL/VTL through `@advice`.

3.4.5.2 Set

The type `Set` represents the mathematical concept of a set. It contains elements without duplicates. Set inherits the operations from `Collection`. For convenience, we added some modifying operations, which may not be supported in certain settings (unmodifiable set), in particular in the context of operation types.

- **setOf(T) - (setOf(T) s)**
  The set difference between *operand* and *s* (i.e., the set of all elements that are in *operand* but not in *s*).
- **T add (T e)**
  Adds the element e to the set *operand*. Does not affect *operand* if *operand* is not modifiable.
- **setOf(T) clone()**
  Returns a cloned shallow copy of *operand*.
- **setOf(A) collect (Expression e)**
  Returns the flattened results of applying the iterator expression *e* to all elements in *operand*.
- **setOf(T) collectNested (Iterator | expression)**
  The container of elements that results from applying *expression* to every member of the *source* set. Nested collections remain in the result.
- **Integer count (T object)**
  The number of times that *object* occurs in the set *operand*.
- **setOf(T) excluding (collectionOf(T) s)**
  Returns a subset of *operand*, which does not include the elements in *s*.
- **setOf(?) flatten()**

[17] "toSet" was the initial operation, "asSet" was added for compatibility with IVML/OCL. "toSet" may be removed in one of the next versions.

Turns *operand* into a flat set. In case that *operand* is already flat and does not contain any sub-collections, this operation does not have any effect. Otherwise the flattened set is returned, i.e., sub-collections are resolved to their containing plain elements.

- **setOf(T) including (collectionOf(T) s)**
  Returns the union of *operand* and *s* (without duplicates).
- **setOf(T) intersection (setOf(T) s)**
  The intersection of *operand* and *s* (i.e., the set of all elements that are in both *operand* and *s*).
- **? iterate (Expression e) / apply(Expression e)**
  Applies *e* to all elements in *operand*. Without aggregating declarator, nothing is returned and just a loop over all elements is executed. If an aggregator is declared, the operation returns a result of the type of the aggregator (denoted as *A* above).
- **T projectSingle()**
  In case of an *operand* with one element, return that element. Otherwise, nothing will be returned and subsequent expressions may fail.
- **setOf(T) reject (Expression e)**
  Returns the elements in *operand*, which do not comply with the iterator *e*.
- **remove (T e)**
  Removes e from *operand* and returns whether *e* was removed. Does not affect *operand* if *operand* is not modifiable.
- **setOf(T) select (Expression e)**
  Returns the elements in *operand*, which comply with the iterator *e*.
- **setOf(Type) selectByKind (Type t) / typeSelect (Type t)**
  Returns all those elements of *operand* that are type compliant with *t* or the subtypes of *t*.
- **setOf(Type) selectByType (Type t)**
  Returns all those elements of *operand* that have the same type as *t*[18].
- **setOf(T) symmetricDifference (setOf(T) s)**
  The symmetric difference between *operand* and *s* (i.e., the set of all elements that are in *operand* or in *s* but not in both).
- **sequenceOf(T) toSequence () / asSequence()**[19]
  Turns *operand* into a sequence.
- **setOf(Type) typeReject (Type t)**
  Returns all those elements of *operand* that are not type compliant with *t* or the subtypes of *t*.
- **setOf(T) union (setOf(T) s)**
  Returns the union of *operand* and *s*.

[18] This operation has been restricted to equality as part of the introduction of selectedByKind in version 0.98.

[19] "toSequence" was the initial operation, "asSequence" was added for compatibility with IVML/OCL. "toSequence" may be removed in one of the next versions.

3.4.5.3 Sequence

A `Sequence` is a `Collection` in which the elements are ordered. An element may be part of a `Sequence` more than once. `Sequence` inherits the operations from `Collection`. For convenience, we added some modifying operations, which may not be supported in certain settings (unmodifiable sequence), in particular in the context of operation types.

- **Boolean ==(collectionOf(Type) c) / equals(collectionOf(Type) c)**
  Returns whether *operand* contains the same elements in the same sequence than *c*. The remaining operations of Any are also available.
- **T [Integer index]**
  The []-operator returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than size().
- **T add (T e)**
  Adds the element e to the set *operand*. Does not affect *operand* if *operand* is not modifiable.
- **sequenceOf(T) append(collectionOf(T) s) / union(collectionOf(T) s)**
  Returns the result of appending *s* to *operand* (possibly including duplicates). Does not affect operand rather than it returns a new sequence.
- **Integer count (T object)**
  The number of times that *object* occurs in the sequence *operand*.
- **sequenceOf(T) append(T object)**
  The sequence consisting of *object*, followed by all elements in *operand*.
- **sequenceOf(T) clone()**
  Returns a cloned shallow copy of *operand*.
- **sequenceOf(A) collect (Expression e)**
  Returns the flattened results of applying the iterator expression *e* to all elements in *operand* in the sequence of *operand*.
- **sequenceOf(T) collectNested (Iterator | expression)**
  The container of elements that results from applying *expression* to every member of the *source* set. Nested collections remain in the result.
- **sequenceOf(Integer) createIntegerSequence(Integer s, Integer e)**
  Creates a sequence of integers from s to e. If e is smaller than s, an empty sequence is created[20].
- **sequenceOf(T) excluding (collectionOf(T) s)**
  Returns a subset of *operand*, which does not include the elements in *s*.
- **T first()**
  Returns the first element of *operand*.
- **sequenceOf(?) flatten()**
  Turns *operand* into a flat set. In case that *operand* is already flat and does not contain any sub-collections, this operation does not have any effect. Otherwise the flattened sequence is returned, i.e., sub-collections are resolved to their containing plain elements.

[20] Please note that this is currently realized by a real enumeration of the values, i.e., extensive sequences shall may not be created. Arbitrary sequences will be targeted in future.

- **T get (Integer index)**
  Returns the element of *operand* at position *index*. *index* must be valid, i.e., not negative and less than size().
- **Boolean hasDuplicates()**
  Returns whether at least one of the elements in *operand* has a duplicate.
- **Integer indexOf(T e)**
  Returns the index of e in *operand*. If e does not exist in *operand*, -1 will be returned.
- **sequenceOf(T) insertAt(Integer index, T object)**
  The sequence consisting of *operand* with *object* inserted at position *index*. Valid indices run from 0 to *size()-1*.
- **? iterate (Expression e) / apply (Expression e)**
  Applies *e* to all elements in *operand*. Without aggregating declarator, nothing is returned and just a loop over all elements is executed. If an aggregator is declared, the operation returns a result of the type of the aggregator (denoted as *A* above).
- **Boolean isSubsequenceOf(sequenceOf(U) o)**
  Returns whether *operand* is a subsequence (considering the sequence and including equality) of *o*.
- **T last()**
  Returns the last element of *operand*.
- **mapOf(T, U) mapAny(sequenceOf(U) o)**
  Maps the elements of *operand* against those in o regardless of their sequence and returns the mapping of equal elements between both.
- **mapOf(T, U) mapSequence(sequenceOf(U) o)**
  Maps the elements of *operand* against those in o in the given sequence and returns the mapping of equal elements between both.
- **Boolean overlaps(sequenceOf(U) o)**
  Returns the sequence that *operand* and *o* have elements in common.
- **sequenceOf(T) prepend(T object)**
  The sequence consisting of *object*, followed by all elements in *operand*.
- **sequenceOf(T) reject (Expression e)**
  Returns the elements in *operand* which do not comply with the iterator expression *e*.
- **remove (T e)**
  Removes the first occurrence of *e* from *operand* and returns whether *e* was removed. Does not affect *operand* if *operand* is not modifiable.
- **removeAll (T e)**
  Removes all occurrences of *e* from *operand*. Does not affect *operand* if *operand* is not modifiable.
- **T removeAt (Integer i)**
  Removes the element at position *i* from *operand*. Does not affect *operand* if *operand* is not modifiable or *i* is outside *operand*'s the valid range. Returns the removed element or undefined.
- **T removeFirst ()**

Removes the first element from *operand*. Does not affect *operand* if *operand* is not modifiable or *operand* is empty. Returns the removed element or undefined.

- **T removeLast ()**
  Removes the last element from *operand*. Does not affect *operand* if *operand* is not modifiable or *operand* is empty. Returns the removed element or undefined.
- **sequenceOf(T) revert() / reverse()**[21]
  Reverts the sequence in *operand* and returns a copy.
- **sequenceOf(T) select (Expression e)**
  Returns the elements in *operand* which comply with the iterator expression *e*.
- **sequenceOf(T) selectByKind (Type t) / typeSelect (Type t)**
  Returns all those elements of *operand* that are type compliant to *t* or the subtypes of *t*.
- **sequenceOf(T) selectByType (Type t)**
  Returns all those elements of *operand* that have the same type as *t*[22].
- **sequenceOf(T) sortAlpha()**
  Sorts the elements in *operand* in alphabetical order using the current locale and the individual string representation of the elements.
- **sequenceOf(T) sortedBy(Expression e) / sort(Expression e)**
  Sorts the elements in *operand* according to the respective values defined by the iterator expression *e*. In case of numerical values, numerical order is considered, else lexicographical order using the default locale.
- **sequenceOf(T) subSequence(Integer l, Integer u)**
  Returns the sub-sequence containing the elements ranging from (0-based) index *l* to *u* of *operand*. If l or u exceed the respective lower or upper bound imposed by the size of operand, *l* or *u* are implicitly corrected, respectively, i.e., assumed to 0 or to *operand*.size() – 1.
- **setOf(T) toSet() / asSet()**[23]
  Turns *operand* into a set (excluding duplicates).
- **sequenceOf(Type) typeReject (Type t)**
  Returns all those elements of *operand* that are not type compliant to *t* or the subtypes of *t*.

No automated conversions are defined among sequences of generic configuration types (e.g., `DecisionVariable`, cf. Section 3.4.7.3) and specific configuration types mapped into VIL/VTL through `@advice`. The `selectByType`/`selectByKind` operations can be used to perform an explicit conversion.

[21] The reverse operation has been introduced as an alias to increase OCL compliance in version 0.98.

[22] This operation has been restricted to equality as part of the introduction of selectedByKind in version 0.98.

[23] "toSet" was the initial operation, "asSet" was added for compatibility with IVML/OCL. "toSet" may be removed in one of the next versions.

3.4.5.4 Map

The `Map` type represents an associative container, which is parameterized over the type of keys `K` and the type of values `V`. For convenience, we added some modifying operations, which may not be supported in certain settings (not modifiable map), in particular in the context of operation types.

- **V [K k] / get (K k)**
  The []-operator returns the value assigned to the given key *k* in *operand*. In case that the mapping does not exist, the expression remains undefined. Use `containsKey` to avoid this.
- **clear()**
  Clears all mappings in *operand*.
- **Boolean containsKey(K k) / hasKey(K k)**
  Returns whether the *operand* contains *k* as key for a mapping.
- **add(K k, V v) / put(K k, V v)**
  Adds the given key-value mapping (*k*,*v*) to the map and overrides existing value for key *k*. Does not affect *operand* if *operand* is not modifiable.
- **V get (K k, V d) / defaultget (K k, V d)**
  Returns the mapping for key *k* in *operand*. If the mapping does not exist, *k* is returned.
- **V getOrAdd (K k, V d)**
  Returns the mapping for key *k* in *operand*. If the mapping does not exist, associate *d* with *k*, i.e., perform *add(k, d)* and return *d*.
- **setOf(K) getKeys() / keys()**
  Returns the keys of *operand* as a non-modifiable set.
- **setOf(K) getValues() / values()**
  Returns the values of *operand* as a non-modifiable set.
- **Boolean isEmpty ()**
  Is the *operand* an empty map?
- **Boolean isNotEmpty () / notEmpty ()**
- Is the *operand* not an empty map?**remove(K k)**
  Removes the mapping for key *k*. Does not affect *operand* if *operand* is not modifiable.
- **sequenceOf(V) sortByKeys(Expression e)**
  Sorts the elements in *operand* according to the sorting of its keys through the iterator expression *e*. In case of numerical values, numerical order is considered, else lexicographical order.
- **Integer size()**
  Returns the size of the *operand* in terms of the number of entries.

Sequences which contain sequences with exactly two entry types matching the types of `Map` can be converted automatically into a `Map` instance.

3.4.5.5 Iterator

`Iterator` is an internal parameterized type that can be used to allow extending types to indirectly return a collection to iterate over, e.g., via map (Section 3.1.9.6)

or for (Section 3.2.9.5). This is useful, if the implementing type shall not or cannot use the VIL types directly.

3.4.6 Version Type

The version type is an internal type (actually not supported as a regular type for variables) for defining version constraints. Using the type name "Version" is discouraged. Thus, the version type supports only the following operations:

- **Boolean == (Version v)**
  Evaluates to true if *operand* and *v* denote the same version.
- **Boolean <> (Any a) / != (Any a)**
  True if the *operand* is different from *a*.
- **Boolean < (Version v)**
  True if the *operand* is less than *v*.
- **Boolean > (Version v)**
  True if the *operand* is greater than *v*.
- **Boolean <= (Version v)**
  True if the *operand* is less than or equal to *v*.
- **Boolean >= (Version v)**
  True if the *operand* is greater than or equal to *v*.

3.4.7 Configuration Types

Configuration types realize the integration with IVML. As the VIL languages are intended for variability instantiation rather than variability modelling, the Configuration types neither support changing the underlying IVML model nor its configuration.

3.4.7.1 IvmlElement

The `IvmlElement` is the most common configuration type. All configuration types discussed in this section are subclasses of `IvmlElement`. Further, this type represents the IVML identifiers used in VIL scripts or templates.

- **Boolean ==(IvmlElement i)**
  Returns whether *operand* is the same IvmlElement as *i*.
- **Annotation getAnnotation (String n) / annotation (String n)**
  Returns the annotation of the *operand* with given name *n*.
- **Annotation getAttribute (String n) / attribute (String n)**[24]
  Returns the annotation of the *operand* with given name *n*.
- **Boolean getBooleanValue () / booleanValue ()**
  Returns the configuration value of the *operand* as a `Boolean`, but is undefined if the underlying IVML decision variable is not of type `Boolean`.
- **Integer getIntegerValue () / integerValue ()**
  Returns the configuration value of the *operand* as an `Integer`, but is undefined if the underlying IVML decision variable is not of type `Integer`.

[24] This is in transition to the new IVML term "annotation". We will keep "attribute" for a certain time and then replace it completely by "annotation".

- **IvmlElement getElement(String n) / element (String n)**
  Returns the (nested) element of the *operand* with given name *n*.
- **EnumValue getEnumValue () / enumValue ()**
  Returns the configuration value of the *operand* as an `EnumValue`, but is undefined if the underlying IVML decision variable is not of type `Enum`.
- **String getName () / name ()**
  Returns the (unqualified) name of the *operand*.
- **Real getRealValue () / realValue ()**
  Returns the configuration value of the *operand* as a `Real`, but is undefined if the underlying IVML decision variable is not of type `Real`.
- **Type getType () / type ()**[25]
  Returns the type of the *operand*.
- **String getQualifiedName () / qualifiedName ()**
  Returns the unqualified name of the *operand*.
- **String getQualifiedType () / qualifiedType ()**
  Returns the unqualified name of the type of the IVML element.
- **String getStringValue () / stringValue ()**
  Returns the configuration value of the *operand* as a `String` in case that the underlying IVML decision variable is of type `String` or the `String` representation of the value in any other case.
- **String getTypeName () / typeName ()**
  Returns the (unqualified) name of the type of the *operand*.
- **Any getValue () / value ()**
  Returns the (untyped) configuration value of the *operand*.
- **Boolean isNull()**
  Returns whether the value of the *operand* is null (in the sense of IVML, see Section 3.3.9).

An `IvmlElement` can automatically be converted to a `String` containing the name of the `IvmlElement`.

3.4.7.2 Enumerations (EnumValue)

`EnumValue` represents an IVML enumeration literal. Values of this type are available independently of any mapping of IVML types into VIL using @advice.

`EnumValue` is a subtype of `IvmlElement` represents an IVML enumeration value. However, it redefines some of the inherited operations as follows

- **EnumValue getEnumValue () / enumValue ()**
  Returns the enum value itself, i.e., *operand*.
- **Integer getIntegerValue () / integerValue ()**
  Returns the (IVML) ordinal of the *operand*.
- **Real getRealValue () / realValue ()**

[25] The getType operation was modified in version 0.98 to enable compliance with OCL. As replacement we introduced the getTypeName operation. Due to the automatic conversion of types into strings, existing scripts relying on getType returning a String can still be used and executed.

Returns the (IVML) ordinal of the *operand* as a `Real`.

- **String getStringValue () / stringValue ()**
  Returns the name of the underlying IVML enum literal of the *operand*.
- **Any getValue () / value ()**
  Returns the enum value itself, i.e., *operand*.

Due to missing semantics, the operations **getBooleanValue () / booleanValue ()** are not available on `EnumValue`. Further, `EnumValue` defines the following operations:

- **Integer getOrdinal () / ordinal ()**
  Returns the (IVML) ordinal of the *operand*.

The specific types defined in IVML models for enumerations and ordered enumerations become available if the underlying IVML model is made accessible to VIL through @advice. If IVML enumerations are mapped via @advice into the IVML type system and their specific operations are available.

Basically, enumeration types provide the same operations as `DecisionVariable` (see Section 3.4.7.3). As in IVML, an ordered enumeration type *E* provides in addition the following operations:

- **Boolean < (E e)**
  True if the *operand* is less than the ordinal of *e*.
- **Boolean > (E e)**
  True if the *operand* is greater than the ordinal of *e*.
- **Boolean <= (E e)**
  True if the *operand* is less than or equal to the ordinal of *e*.
- **Boolean >= (E e)**
  True if the *operand* is greater than or equal to the ordinal of *e*.
- **E max (E e)**
  The maximum value of the *operand* and *e* w.r.t. their ordinals.
- **E min (E e)**
  The minimum value of the *operand* and *e* w.r.t. their ordinals.

3.4.7.3 DecisionVariable

This subtype of `IvmlElement` represents a configured IVML decision variable. Decision variables that have not been configured / frozen are not accessible and containing expressions cannot be evaluated (will be ignored).

- **sequenceOf(DecisionVariable) variables()**
  Returns the frozen nested variables of *operand*. Except for the IVML types container and compound, this sequence will always be empty. The result cannot be modified.
- **DecisionVariable getByName(String name) / byName(String name)**
  Returns the decision variable in *operand* (if it exists).
- **IvmlDeclaration getDeclaration() / declaration()**
  Returns the declaration of the decision variable in *operand*.
- **String getVarName() / varName()**

Returns the name of the underlying (dereferenced) decision variable of the *operand*. In case of a reference, getName()/name() will return the name of the reference variable but not of the referenced variable.

- **sequenceOf(Annotation) annotations() / attributes()**
  Returns the frozen annotations of *operand*. Except for the IVML types container and compound, this sequence will always be empty. The result cannot be modified.
- **setValue(Any v)**
  Changes the value of the underlying decision variable of the *operand* by setting the new value *v*. Attempts to change a frozen variable or to set an incompatible value will be ignored.
- **createValue(Boolean o)**
  Creates a new value of the same (declared) type as *operand* and assigns it as value to *operand*. This operation overrides an existing value if *o* evaluates to true. Attempts to change a frozen variable will be ignored.
- **createValue(Type t, Boolean o**
- Creates a new value of the given type *t* as *operand* and assigns it as value to *operand*. This operation overrides an existing value if *o* evaluates to true. Attempts to assign an incompatible value (*t* does not comply with operand) or to change a frozen variable will be ignored.
- **Boolean isFrozen()**
  Returns whether the underlying (dereferenced) decision variable of the *operand* is frozen and cannot be changed.
- **Boolean isConfigured() / isDefined()**
  Returns whether the underlying (dereferenced) decision variable of the *operand* is configured. Please note that `isNull()` implies `isConfigured()`, as the null of IVML configures a decision variable.
- **Boolean hasConfiguredValue() / hasDefinedValue()**
  Returns whether the underlying (dereferenced) decision variable of the *operand* `isConfigured()` and not `isNull()`.
- **DecisionVariable getParent() / parent()**
  Returns the parent variable of *operand* if available. If operand is a top-level variable, i.e., the actual parent is a configuration, the result is undefined.
- **Configuration selectAll()**
  Returns a configuration as a projection of the nested decision variables in *operand* also returned by `variables()` but in terms of a configuration.

A `DecisionVariable` can automatically be converted into `Integer`, `Real`, `Boolean`, `String` or `EnumValue` in terms of its respective value and type. Further, a `DecisionVariable` can automatically be converted into a sequence of decision variables using the `variables()` operation in order to simplify loop declarations.

3.4.7.4 Annotation[26]

This subtype of `IvmlElement` represents a configured IVML annotation. This subtype does not specify any additional operations over `DecisionVariable`.

An `Annotation` can automatically be converted into `Integer`, `Real`, `Boolean`, `String` or `EnumValue` in terms of its respective value and type. Further, an `Annotation` can automatically be converted into a sequence of decision variables using the `variables()` operation in order to simplify loop declarations.

3.4.7.5 IvmlDeclaration

This subtype of `IvmlElement` represents the type underlying an IVML decision variable. Instances of this type do not provide any configuration values rather than providing access to the structure of the represented type, e.g., the nested variable declarations in an IVML compound.

3.4.7.6 Configuration

The `Configuration` type provides access to configuration values as well as to the type declarations for a certain IVML model. In particular, the configuration type allows to create projections of a configuration, e.g., to select a subset of decision variables and specify further operations on that subset such as passing it to rules or to (one or more) instantiators. Although being an `IvmlElement`, a configuration instance will not provide access to values.

In traditional product line instantiations, the Configuration will contain *frozen* variables only, i.e., variables that are explicitly intended for instantiation. As VIL is also intended to support runtime variability, i.e., instantiation at runtime in the sense of Dynamic Software Product Lines (DSPL) [10, 11], Configurations may be re-reasoned or changed. We will list specific operations intended for DSPL instantiations in a separate operation list below in this section.

- **sequenceOf(DecisionVariable) variables()**
  Returns the configured and frozen decision variables of *operand*.
- **sequenceOf(Annotation) annotations() / attributes()**
  Returns the configured and frozen annotations of *operand*.
- **Boolean isEmpty()**
  Returns whether configuration in *operand* is empty, i.e., does not contain decision variables. This may occur due to the projection capabilities of this type.
- **DecisionVariable getByName(String n) / byName(String n)**
  Returns the specified decision variable (if it exists).
- **Configuration selectByName(String n)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression *n* applied on (qualified and unqualified) decision variable names.
- **Configuration selectByType(String t)**

[26] This is in transition to the new IVML term “annotation”. We will keep “attribute” for a certain time and then replace it completely by “annotation”.

Returns a configuration as a projection of *operand* containing those decision variables, which are of the type specified in terms of a Java regular expression *t* applied on (qualified and unqualified) type names.

- **Configuration selectByAnnotation(String n) / selectByAttribute(String n)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are annotated by the annotation specified in terms of a Java regular expression *n* applied on annotation names.
- **Configuration selectByAnnotation(String n, Any v) / selectByAttribute(String n, Any v)**
  Returns a configuration as a projection of *operand* containing those decision variables which are annotated by the specified annotation (in terms of an IVML identifier *n*) and value *v*.
- **Configuration selectByProject(String n)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are defined by the project given by its name *n* and its imported projects. If the project cannot be found, an empty configuration is returned.
- **FileArtifact store(Path p)**
  Stores the underlying (unprojected) configuration of *operand* into *p* containing user-defined configuration values only.
- **FileArtifact store(Path p, Boolean u)**
  Stores the underlying (unprojected) configuration of *operand* into *p* containing user-defined configuration values (if *u* is true) or all configuration values including automatically derived ones (if *u* is false).
- **Configuration selectByProject(String n, Boolean i)**
  Returns a configuration as a projection of *operand* containing those decision variables, which are defined by the project given by its name *n* and, depending whether *i* is true, its imported projects. If the project cannot be found, an empty configuration is returned.
- **setOf(?) allInstances (Type t)**
  Returns all instances of type *t* in the *operand*. If no instances are known, an empty set is returned. The actual type of the returned set complies to the underlying type, in particular IVML compound types mapped via @advice into IVML.

3.4.8 Built-in Artifact Types and Artifact-related Types

In this section, we will discuss the built-in artifact types as well as artifact-related types such as paths. Please note that the (meta model) of the artifact model is extensible, so that further as well as derived types may be added if needed.

3.4.8.1 Path

A path represents a relative or absolute file or folder. Paths are always relative to the containing project, in more detail to the containing artifact model and normalized in Unix notation (using the slash as path separator). Further, paths may be patterns and contain wildcards according to the ANT conventions [9].

- **String getPath() / path()**

Returns a string representation of *operand*.

- **Boolean isPattern()**
  Returns whether the *operand* is a pattern, i.e., whether it contains wildcards.
- **JavaPath toJavaPath()**
  Explicitly converts the *operand* into a Java package path.
- **JavaPath toJavaPath(String p)**
  Explicitly converts the *operand* into a Java package path and removes the prefix Java regular expression *p* (might be due to src folders in Eclipse or Maven).
- **String toOSPath()**
  Turns the *operand* into a relative operating system specific path.
- **String toOSAbsolutePath()**
  Turns the *operand* into an absolute operating system specific path.
- **deleteAll()**
  Deletes all artifacts in the *operand* path. Use this function to delete pattern paths.
- **delete()**
  Deletes the artifact underlying the *operand*. This operation will cause no effects on pattern paths.
- **mkdir()**
  Creates a directory from the *operand* path. This operation will fail if applied to a pattern.
- **touch()**
  Changes the last modification date of the underlying filesystem element to now. If the path denotes a file and the file does not exist, creates the file.
- **setOf(FileSystemArtifact) copy (FileSystemArtifact t)**
  Copies all file system artifacts denoted by the *operand* to the location of *t* and returns the created artifacts at the target location.
- **setOf(FileSystemArtifact) move (FileSystemArtifact t)**
  Moves all file system artifacts denoted by the *operand* o the location of *t* and returns the created artifacts at the target location.
- **Boolean matches(Path p)**
  Returns whether the (pattern in) *operand* matches the given path.
- **setOf(FileArtifact) selectByType(Type t)**
  Returns those artifacts matching *operand* and that have the same type as *t*[27].
- **setOf(FileArtifact) selectByKind(Type t) / setOf(FileArtifact) typeSelect(Type t)**
  Returns those artifacts matching *operand* and complying with the given artifact type *t* including subtypes of *t*.
- **setOf(FileArtifact) typeReject(Type t)**
  Returns those artifacts matching *operand* and not complying with the given artifact type *t* including subtypes of *t*.
- **setOf(FileArtifact) selectAll()**
  Returns all artifacts matching *operand*.

[27] This operation has been restricted to equality as part of the introduction of selectedByKind in version 0.98.

- **String +(String s)**
  Returns the string concatenation of *operand* and the given String *s*.
- **Boolean exists()**
  Returns whether the artifact denoted by the path exists. The operation will always return false in case of a pattern path.
- **String getName() / name()**
  Returns the name of the file part of the path or, in case of a pattern path, the entire pattern path.
- **String getMd5Hash()**
  Returns the MD5 hash of the file system artifact in *operand*. May fail if the underlying file is not accessible or the MD5 algorithm is not available.
- **Path rename(String n)**
  Renames the *operand* to *n* and returns the related new path. An unqualified name *n* is interpreted in the context of the respective artifact, e.g., for a file artifact the own path is prepended. In general, qualified names are encouraged if possible.

Paths can be constructed from a `String` using the explicit constructor.

Conversions: A `Path` can be converted automatically into a `FolderArtifact`.

3.4.8.2 JavaPath

A subtype of `Path` representing Java package paths (separated by ".").

- **String getPathSegments() / pathSegments()**
  The path segments of *operand* without the file name.
- **JavaPath getPathSegmentsPath() / pathSegmentsPath()**
  The path segments of *operand* without the file name.

Conversions: A `String` can be converted automatically into a `JavaPath`.

3.4.8.3 ProjectSettings

The type `ProjectSettings` is an internal super type of all keys that can be used to specify global project settings. Currently, only the Java language extension defined in the VIL/VTL extensions document defines a key for the classpath.

3.4.8.4 Project

The project type encapsulates the physical location of a product line including all artifacts, in particular in terms of an Eclipse project. There are no explicit constructors for this type, as instances will be provided by the VIL/EASy-Producer runtime environment.

- **String getName() / name()**
  Returns the name of the project in *operand*.
- **String getPlainName() / plainName()**
  The name of *operand* without file name extension.
- **setOf(FileArtifact) selectAllFiles()**
  Returns all file artifacts in *operand*.
- **setOf(FileArtifact) selectAllFolders()**

Returns all folder artifacts in *operand*.

- **setOf(Project) predecessors()**
  Returns all predecessor projects in *operand*. In standalone product lines, the result may be empty. In hierarchical product lines, the predecessors are returned.
- **Path getPath() / path()**
  Returns the (base) path the *operand* is located in.
- **Path localize(Project s, Path p)**
  Localizes path *p* originally in project s to the project in *operand*.
- **Path localize(Project s, FileSystemArtifact a)**
  Localizes path of *a* originally in project s to the project in *operand*. This operation does neither copy nor move *a*.
- **setOf(FileArtifact) selectByType(Type t)**
  Returns those file artifacts in the *operand* project that have the same type as *t*.
- **setOf(FileArtifact) selectByName(String r)**
  Returns those file artifacts in the *operand* project for which their name complies with the Java regular expression in *r*.
- **Path getEasyFolder() / easyFolder()**
  Returns the path to the EASy producer configuration files in *operand*.
- **Path getIvmlFolder() / ivmlFolder()**
  Returns the path to the IVML models in *operand* (typically the same as `getEasyFolder()`).
- **Path getVilFolder() / vilFolder()**
  Returns the path to the VIL build specification models in *operand*.
- **Path getVtlFolder() / vtlFolder()**
  Returns the path to the VTL template models in *operand*.
- **setSettings(ProjectSettings key, Any object)**
  Sets the settings for the artifact model
- **Any getSettings(ProjectSettings key)**
  Returns the settings object for the specified key

Conversions: A `Project` can be converted automatically into its base `Path`.

3.4.8.5 Text

Represents an artifact or a fragment artifact in terms of the underlying text and allows direct manipulations. The manipulations will be written back into the artifact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artifact.

- **Boolean isEmpty()**
  Returns whether the text representation in *operand* is empty.
- **String getText() / text()**
  Returns the textual contained in the text representation in *operand*. Please note that the result is disconnected from the textual representation, i.e., changes made on operand will not be reflected into the result and vice versa.

- **setText(String t)**
  Replaces the contents of the text representation in *operand* by t.
- **Boolean matches(String regEx)**
  Returns whether the specified *regEx* matches the textual representation in *operand*.
- **Text substitute(String regEx, String r)**
  Substitutes all occurrences of *regEx* in *operand* by *r* and returns the modified text.
- **Text replace(String s, String r)**
  Substitutes all occurrences of *s* in *operand* by *r* and returns the modified text. This operation does not consider regular expression matches rather than direct text matches.
- **Text append(String s)**
  Appends *s* to the end of *operand* and returns the modified text.
- **Text prepend(String s)**
  Prepends *s* before the beginning of *operand* and returns the modified text.
- **Text append(Text t)**
  Appends the entire contents of *t* to the end of *operand* and returns the modified text.
- **Text prepend(Text s)**
  Prepends the entire contents of *t* before the beginning of *operand* and returns the modified text.

Conversions: Please note that a text representation cannot be converted automatically into a String. This shall avoid confusion as, in contrast to a text representation, the resulting String is disconnected from the underlying artifact and changes to the String will not affect the artifact.

3.4.8.6 Binary

Represents an artifact or a fragment artifact in terms of the underlying binary form and allows direct manipulations. This type is subject to future extensions. The manipulations will be written back into the artifact at the end of the lifetime of the related VIL variable. A text representation may be modifiable or read-only depending on the underlying artifact.

- **Boolean isEmpty ()**
  Returns whether the binary representation in *operand* is empty.

3.4.8.7 Artifact

This is the most common artifact type. All specific artifact types are subtypes of `Artifact`.

- **delete()**
  Delete the artifact in *operand* regardless of whether it is a file, a component, or a fragment within an artifact.
- **Boolean exists ()**

Returns whether the artifact in *operand* actually exists. Please note that the semantics of this operation depends on the type of artifact, e.g., for a FileArtifact this operation returns whether there is an actual underlying file.

- **String getName () / name()**
  Returns the name of the artifact in *operand*. The specific meaning of the name depends on the actual artifact type.
- **Text getText() / text()**
  Returns the textual representation of the artifact in *operand*. Whether the representation can be manipulated depends on whether the artifact itself may be modified.
- **Binary getBinary() / binary()**
  Returns the binary representation of the artifact in *operand*. Whether the representation can be manipulated depends on whether the artifact itself may be modified.
- **rename(String n)**
  Renames the artifact in *operand*. The specific effect of this operation and whether it may be applied at all depends on the actual artifact type.

3.4.8.8 FileSystemArtifact

Represents the most common type of file system artifacts.

- **Path getPath() / path()**
  Returns the path to *operand*.
- **setExecutable(Boolean o)**
  Tries to set an operation system specific flag to allow for the execution of the artefact in *operand*. *o* indicates, if only the owner shall be able to execute the artefact.
- **setOf(FileSystemArtifact) move(FileSystemArtifact a)**
  Moves the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.
- **setOf(FileSystemArtifact) copy(FileSystemArtifact a)**
  Copies the artifact in *operand* to the location of *a*. If *a* exists, it will be overwritten.

Conversions: A `FileSystemArtifact` can be converted automatically into its `Path`.

3.4.8.9 FolderArtifact

This type represents a folder in the file system and always belongs to a certain artifact model (and typically to a containing `Project`). `FolderArtifact` is a subtype of `FileSystemArtifact`.

- **setOf(FileArtifact) selectAll()**
  Returns all file system artifacts contained in *operand*.

Conversions: `FolderArtifact` can automatically be converted into a `String` containing the path or into a `Path` denoting the location.

3.4.8.10 FileArtifact

This type represents a file in the file system and always belongs to a certain artifact model (and typically to a containing `Project`). `FileArtifact` is a subtype of `FileSystemArtifact`. Please note that the actual instance in a variable of type `FileArtifact` may belong to a subtype as the creation of artifact takes artifact specific rules into account.

A temporary `FileArtifact` can be constructed using the constructor without arguments. A string (containing a path) as well as a `Path` can be converted automatically into a `FileArtifact`.

A `FileArtifact` is created as the default fallback, i.e., if no more specific artifact matches the underlying real artifact. The artifact creation mechanism may be configured using an external Java properties file, which relates artifact names to file path patterns (overwriting or extending the built-in rules).

- **String getMd5Hash()**
  Returns the MD5 hash of the file system artifact in *operand*. May fail if the underlying file is not accessible or the MD5 algorithm is not available.
- **Boolean hasSameContent(FileArtifact f)**
  Returns whether the file system artifact in *operand* has the same byte-wise content as *f*.

3.4.8.11 VtlFileArtifact

The `VtlFileArtifact` type is a subtype of `FileArtifact`. In particular, it helps the VTL template instantiator in dynamic dispatch between other types of file artifacts and actual VTL templates. It does not provide additional operations or conversions over the `FileArtifact`.

A `VtlFileArtifact` is created for all real file artifacts with file extension `vtl`.

3.4.8.12 XmlFileArtifact

The `XmlFileArtifact` is a built-in composite artifact, i.e., if it exists it is either empty or contains a valid XML document. Its content is analysed for known substructures, which are made available for querying and manipulation in terms of fragment artifacts. One aim of the XML artifact implementation is to provide control over the sequence of elements and attributes, i.e., manipulations of existing XML files shall have minimum impact and creation of XML files shall exactly follow the sequence of commands in VIL/VTL. This default behaviour can be switched off. Please see also the warnings regarding text operations below.

- **XmlFileArtifact XmlFileArtifact()**
  Creates a temporary XML file artifact. Please note that the file artifact is initially empty and needs the creation of a root element (see below).
- **XmlElement getRootElement() / rootElement()**
  Returns the root element of the XML artifact in *operand*.
- **setOf(XmlElement) selectAll()**
  Returns all XML elements contained in *operand*.
- **setOf(XmlElement) selectChilds()**

Returns all XML elements contained in the root element of *operand*.

- **setOf(XmlAttribute) selectByName (String r)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression.
- **setOf(XmlAttribute) selectByPath (String p)**
  Returns those XML elements in *operand*, which comply with the given name path *p*. Here, a path is a hierarchical name separated by slashes denoting the names of nested elements (except for the root element which is implicitly selected). Path elements may be Java regular (sub-)expressions.
- **setOf(XmlAttribute) selectByXPath (String p)**
  Returns those XML elements in *operand*, which comply with the given XPath[28] *p*. As *p* is a String, VIL variable and expression replacements happen before evaluating of the path.
- **XmlElement createRootElement(String n)**
  Creates the root element of the *operand* with name *n* if no root element exists. Otherwise, returns the root element.
- **XmlElement createElement(String n, String c="")**
  Creates a new XmlElement with name *n* as child of the operator element *p* with optional initial cData content *c* and returns the new XmlElement.
- **setOf(XmlAttribute) selectByRegEx (String r)**
  Returns those XML elements in *operand*, which comply regarding their name with the given Java regular expression *r*.
- **setIndentation (Integer i)**
  Defines the per-line indentation in terms of *i* whitespaces used while storing/formatting this artifact. A negative value of *i* disables indentation. The default indentation is 4 whitespaces.
- **setEncoding(String e)**
  Defines the encoding of the *operand* document.
- **String getEncoding()**
  Returns the encoding of the *operand* document.
- **setXmlStandalone(Boolean s)**
  Defines the XML standalone flag of *operand*.
- **Boolean getXmlStandalone()**
  Returns the XML standalone flag of *operand*.
- **setOmitXmlDeclaration(Boolean o)**
  Disables or enables emitting the XML declaration header, i.e., the `<?xml` part for the *operand* artifact. By default, the declaration header is emitted.
- **setOmitXmlStandalone(Boolean o)**
  Disables or enables emitting the Standalone attribute in the XML declaration header of the *operand* artifact. By default, the standalone attribute is omitted if it has not value true.
- **setSynchronizeAttributeSequence(Boolean s)**
  Disables or enables synchronization of the XML attributes assigned to an XML element between artifact and XML file. By default, synchronization is

[28] http://www.w3.org/TR/xpath20/

enabled, i.e., attributes are emitted exactly in the way they were read / created. If disabled, the implementation determines the sequence of the attributes, currently (Xalan) by sorting the attributes according to their name in ascending literal oder.

- **format()**
  Formats the *operand* artifact based on the currently available content. If VTL content statements are used to create the *operand* artifact, the format operation shall be called from VIL as VTL automatically persists the content at the end of the script.

A `XmlFileArtifact` is created for all real file artifacts with file extension `xml`. Automated conversion is supported from `String` and `FileSystemArtifact`.

Please be aware of the fact that appending to a `XmlFileArtifact` via the textual representation (although available) may lead to illegal XML documents, and, as the `XmlFileArtifact` works on valid XML documents only, appending operations may be ignored. Further, appending to an empty `XmlFileArtifact` leads to a default valid XML document that potentially does not include the appended elements. In general, we recommend using the operations of the specific XML types. However, if you would like to rely on the textual representation, please either use valid replacements or change the entire text by setting a valid XML string.

XmlNode

The `XmlNode` type is an abstract built-in fragment artifact, which represents any kind of node within a `XmlFileArtifact`. In particular, it inherits all operations from `Artifact` such as access to the representations of the contained text and adds XML specific operations.

- **Text getCdata() / Cdata()**
  Returns the contents / CDATA information of *operand* as a textual representation.
- **setCdata(String c)**
  Changes the contents / CData information of *operand* by overriding it with the contents of *c*.
- **XmlElement getParent() / parent()**
  Returns the parent element of *operand*.

XmlElement

The `XmlElement` is a built-in fragment artifact specializing `XmlNode`. In particular, it inherits all operations from `XmlNode` such as access to the representations of the contained text or CDATA. Please note that deleting the root element of an XML document is not supported due to technical restrictions. However, modifying the root element or deleting and recreating the entire artifact is possible (see Section 4.3.5).

- **XmlElement XmlElement(XmlElement p, String n, String c="")**
  Creates a new XmlElement with name *n* as child of the parent element *p* with optional initial cData content *c* and returns the new XmlElement.
- **XmlElement XmlElement(XmlFileArtifact p, String n, String c="")**

Creates a new XmlElement with name *n* as child of the root element of XMLFileArtifact *p* with optional initial cData content *c* and returns the new XmlElement.

- **XmlElement buildElement(XmlElement p, String n, String c="")**
  Creates a new XmlElement with name *n* as child of the parent element *p* with optional initial cData content *c* and returns *p* (for chaining).
- **XmlAttribute buildAttribute(XmlElement p, String n, String v, Boolean f=true)**
  Creates a new XML attribute for in *p* with name *n* and value *v*. In case that an equally named attribute already exists in p, the attribute is overwritten if *f* is true or created anyway (*f* is false) and returns *p* (for chaining).
- **XmlElement createElement(String n, String c="")**
  Creates a new XmlElement with name *n* as child of the operator element *p* with optional initial cData content *c* and returns the new XmlElement.
- **setOf(XmlElement) selectAll()**
  Returns all XML elements contained in *operand*.
- **sequenceOf(XmlNode) nodes()**
  Returns all XML child nodes nested in *operand* in definition sequence.
- **sequenceOf(XmlElement) elements()**
  Returns all XML element nodes nested in *operand* in definition sequence.
- **sequenceOf(XmlComment) comments()**
  Returns all XML comment nodes nested in *operand* in definition sequence.
- **setOf(XmlAttributes) attributes()**
  Returns all XML attributes belonging to *operand*.
- **XmlAttributes getAttribute(String n)**
  Returns the XML attribute in *operand* with the specified name *n*. Please note that this operation does not fail, if the specified attribute does not exist but rather the subsequent evaluation will stop.
- **XmlAttributes setAttribute(String n, String v)**
  Defines or changes the XML attribute in *operand* with the specified name *n* to the given value *v*.
- **setOf(XmlAttribute) selectByName (String r)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression.
- **setOf(XmlAttribute) selectByName (String r, Boolean c)**
  Returns those XML elements in *operand*, which comply with the given name pattern specified as Java regular expression. The parameter *c* denotes whether the name comparison shall consider case sensitivity.
- **setOf(XmlAttribute) selectByRegEx (String r)**
  Returns those XML elements in *operand*, which comply regarding their name with the given Java regular expression *r*.
- **setOf(XmlAttribute) selectByPath (String path)**
  Returns those XML elements in *operand*, which comply with the given name path. Here, a path is a hierarchical name separated by slashes denoting the names of nested elements (except for the root element which is implicitly selected). Path elements may be Java regular (sub-)expressions.

- **setOf(XmlAttribute) selectByXPath (String p)**
  Returns those XML elements in *operand*, which comply with the given XPath[28] *p*. As *p* is a String, VIL variable and expression replacements happen before evaluating of the path.

XmlComment

The `XmlElement` is a built-in fragment artifact specializing `XmlNode`. In particular, it inherits all operations from `XmlNode` such as access to the representations of the contained text or CDATA. Currently, only XML comments within the document root node are supported.

XmlAttribute

The `XmlElement` is a built-in fragment artifact, which belongs to the `XmlFileArtifact` and to the fragment artifact `XmlElement`. In particular, it inherits all operations from `Artifact`.

- **XmlAttribute XmlAttribute(XmlElement p, String n, String v, Boolean f=true)**
  Creates a new XML attribute for in *p* with name *n* and value *v* and returns the new XmlAttribute. In case that an equally named attribute already exists in p, the attribute is overwritten if *f* is true or created anyway (*f* is false) .
- **XmlElement getParent() / parent()**
  Returns the parent element of *operand*.
- **String getValue () / value()**
  Returns the value of the attribute in *operand*.
- **setValue (String v)**
  Changes the value of the attribute in *operand* to *v*.

3.4.9 Built-in Instantiators

VIL provides also several built-in instantiators, in particular to modify or generate entire artifacts in one step. Basically, instantiators shall be defined using the VIL template language (this actually happens through an instantiator for VIL templates). However, very complex instantiation operations as well as existing (legacy) instantiator operations shall be available to VIL, also as a better integrated alternative to just calling an operating system command. In this section, we will discuss the instantiators shipped with the VIL implementation as well as their specific operations. Please refer to the EASy-Producer developer documentation on how to realize custom instantiators.

3.4.9.1 VIL Template Processor

The VIL template processor is responsible for interpreting and executing VIL template scripts in close collaboration with VIL. It may operate in two different modes depending on the actual arguments, namely executing VIL templates or replacing VIL expressions in a file artifact.

Basically, VTL receive three different parameters, the template, the variability configuration and the target artifact (fragment) to be produced. Thereby, the instantiator itself takes an argument of type `Artifact`, but the dynamic dispatch

mechanism allows specifying subtypes in the template parameters or even to have multiple main subtemplates for different artifact types. In addition the VIL template processor may receive an arbitrary number of named arguments specific to the template to be executed.

This instantiator provides three instantiator operations:

- **setOf(FileArtifact) vilTemplateProcessor(String n, Configuration c, Artifact a, Boolean v=false, …)**
- Parses and analyses the template given by its name *n* (version restriction see Section 3.1.4), executes the template specification using the configuration *c* and replaces the content of *a*. If *v* is true, add an implicit advice to the IVML project represented by *c* so that IVML types can be resolved and more convenient expressions are possible. Additional named arguments are passed to the VTL template *t* in order to customize the processing and must comply to the additional template parameters.
- **setOf(FileArtifact) vilTemplateProcessor(VtlFileArtifact t, Configuration c, Artifact a, …)**
  Parses and analyses the template in *t*, executes the template specification using the configuration *c* and replaces the content of *a*. Additional named arguments are passed to the VTL template *t* in order to customize the processing and must comply to the additional template parameters.
- **setOf(FileArtifact) vilTemplateProcessor(FileArtifact t, Configuration c, Artifact a, Boolean v=false)**
  Searches for VIL variables and expressions in *t* (variables given as `$variableName`, expressions as `${expression}`) and replaces them if specified in nested/recursive manner by their actual value as defined in the configuration *c*. The output replaces the content of *a*. If *v* is true, add an implicit advice to the IVML project represented by *c* so that IVML types can be resolved and more convenient expressions are possible.

If a VIL template is passed to the template processor, the template is executed as specified. If an ordinary file is passed to the template processor, the file itself is considered as template (no template header is needed). In this mode, VIL expressions given as variable references, expressions and in-content statements (see Section 3.2.9.7) are replaced by actual values in the text representation of the artifact. Moreover, the VIL template processor allows defining variables to ease the application of the other in-content commands. In summary, the VIL template processor supports:

- Variable reference:
  $*var*
  var is resolved to a variable in the current context and implicitly the configuration passed into the VIL template processor. The whole in-content statement is replaced by the value of *var*. The passed-in configuration is available as pre-defined variable *config*. Qualified IVML names denoting IVML variables separated by *::* can be used here.
- Expression reference:
  ${*expr***}**

expr is resolved to a VTL expression, evaluated and the result replaces the whole in-content statement. This form includes variables, i.e., **${***var*} is the same as **$var.** The passed-in configuration is available as pre-defined variable *config*. Nested expressions, e.g., within a String parameter of a function are considered and replaced accordingly.

- Variable declaration:
  ${VAR *var = init***}**
  Declares the template-local variable *var* and initializes it with *init*. The type of *var* is the dynamic type of *init*. If declared once, the value of *var* can currently not be re-assigned. *var* can be used in other in-content statements. The entire in-content statement is removed from the result. Please note that this in-content statement is only available in file-templates, not in VTL templates (usual variable declarations apply there).
- Alternative:
  ${IF** *condition***}***then-part***${ELSE}*else-part***${ENDIF}**
  The matching alternative is emitted, the other alternative disappears (as well as the in-content markers). For details, see Section 3.2.9.7.
- Loop:
  ${FOR** *var* **:** *init* **SEPARATOR** *ex* ***END*** *ex***}***body-part***${ENDFOR}
  Replaces the entire content statement by an iterative evaluation of the *body-part*. Within *body-part*, each occurrence of *var* is replaced by the actual value taken from init (which must evaluate to a container value). For further details, see Section 3.2.9.7. Iterations over IVML container collections and *body-part* expressions accessing container slots may require type resolution, i.e., the optional parameter *v* of the template processor shall have the value true.
- Import:
  ${IMPORT *name* **WITH** *ex***}**
  Imports a VTL script with given *name*. As for VIL/VTL exports, a version restriction expression can be given in the optional part **WITH** *ex*. Please note that import in-contents commands can be given in any position of the template file and make imported VTL defs only available for all following in-content commands. The entire in-content statement is removed from the result.

The in-content statements above may be escaped by prefixing a backslash, i.e., the in-content statement is ignored and emitted as it is without the escaping backslash.

In addition, similar operations are provided which take a **collection of artifacts** as input. At a glance, storing the output produced by the same template in multiple files might be superfluous but it simplifies the application of the VTL template processor in conjunction with operations which return a collection with one element, such as copying a file (without further utilizing the `projectSingle` operation).

3.4.9.2 ZIP File Instantiator

The ZIP file blackbox instantiator allows packing and unpacking ZIP files.

- **setOf(FileArtifact) zip(Path b, Path a, Path z)**

Packs the artifacts denoted by the path *a* (possibly a path pattern) into the ZIP file denoted by path *z* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting ZIP file. If *z* does not exist, a new artifact is created. If *z* exists, the contents of *z* is taken over into the result ZIP, i.e., the artifacts in *a* are added to *z*.

- **setOf(FileArtifact) zip(Path b, collectionOf(FileArtifact) a, Path z)**
  Packs the artifacts in *a* into the ZIP file denoted by path *z* while taking path *b* (or a project) as a basis for making the file names of the artifacts relative in the resulting ZIP file. If *z* does not exist, a new artifact is created. If *z* exists, the contents of *z* is taken over into the result ZIP, i.e., the artifacts in *a* are added to *z*.
- **setOf(FileArtifact) unzip(Path z, Path t)**
  Unpacks artifacts in the ZIP file *z* into the target path *t*.
- **setOf(FileArtifact) unzip(Path z, Path t, String p)**
  Unpacks those artifacts in the ZIP file *z* matching the ANT pattern *p* into the target path *t*.

4 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 sections above, we will discuss a collection of typical application patterns 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. For artifact-specific how-to hints, please refer to the VIL extension documentation [16].

4.1 VIL

4.1.1 Copy Multiple Files

One recurring task is to copy multiple files, frequently from the source to the target project in order to prepare instantiation. Basically, multiple files can be described by a regular path expression in ANT notation. Let's assume that we want to copy all Java files in the `src` folder of the source project to the `src` folder of the target project:

The hard way

Copying all those files can be described in imperative fashion as follows:

```
Path p = "$source/src/**/*.java";
map(f = p.selectAll()) {
  copy(f, "$target" + p.path().substring("$source".length());
}
```

However, this needs abusing a map as a loop and manually taking care of the target artifact names and it does not care whether files actually need to be copied.

The path copy operation

In order to simplify the copy fragment above, we can just use the related path operation:

```
Path p = "$source/src/**/*.java";
copy(p, $target); // or p.copy($target) if you like
```

Although this is much simpler, it still copies all files.

A bit smarter

Instead of imperative coding, we rely on VIL rules:

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

In this fragment, the VIL pattern matching algorithm takes care of relating source and target artifacts and copies only files if needed, i.e., the target does not exist or is outdated. However, for each pair of patterns you need a specific rule.

Smart and reusable

To make the copy rule shown above reusable, we introduce a parameter:

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

4.1.2 Convenient Shortcuts

Sometimes selection or artifact operations lead to exactly one element as it is intended by the script designer due to the structure of the variability model or the realization of the product line. However, these operations return a collection instead of a single value so that `sequence[0]`, `sequence.get(0)`, `sequence.first()`, `set.asSequence().get(0)` etc. must be applied to obtain that single instance.

In case of sets, this can be simplified by `set.projectSingle()`. Further, if the instance shall further be processed by an instantiator or the VIL template processor, frequently also a collection can be passed in instead of a single artifact so that the operations shown above are not required at all, e.g.,

```
vilTemplateProcessor(”template.vtl”, config, set);
```

4.1.3 Projected Configurations

Frequently, templates or the velocity instantiator do not need access to the full configuration. Basically, passing in subsets of a configuration speeds up the instantiation process. However, in some cases, it is even required to work on a subset of the configuration, e.g., to select a certain element of an IVML container and to continue on the decision variables of that element, e.g., a compound. Consider the following IVML fragment

```
Compound Workflow {
  String name;
  Boolean enabled;
}
sequenceOf(Workflow) workflows;
```

Here, the user may configure different workflows the resulting product shall provide. While generating the workflow implementation (bindings) with VIL or their configuration, typically each configured workflow is processed and the values are taken over. If this shall be realized with velocity, the velocity processor does not know which workflow actually shall be processed. A simple solution in VIL is

```
map(wf = config.selectByName(”workflows”).variables) {
  String wfName = wf.selectByName(”name”).stringValue();
  copy(wfTemplate, ”$target/src/workflows/$wfName.java”);
  velocity(wfFile, wf.selectAll());
}
```

This fragment processes all compound instances in workflows, prepares the instantiation of each workflow by copying the workflow template `wfTemplate` to the right location and by finally by instantiating the template using velocity. Thereby, the nested decision variables from the actual workflow `wf` are projected into a

configuration (`wf.selectAll()`) and passed to velocity where then `$name` and `$enabled` can directly be used as placeholders for the actual values.

4.2 VIL Template Language

In this section we will discuss some patterns for the VIL template language.

4.2.1 Don't fear named parameters

Basically, a VIL template takes two parameters, the configuration and the target artifact. In many situations, already further parameter are helpful, just to parameterize the template or to pass already determined information (from artifacts, the configuration or both) into the template and to simplify the template. Therefore, a VIL template may take an arbitrary number of named parameters.

```
template properties(Configuration config, FileArtifact target,
  String name) {

  def main(Configuration config, FileArtifact target, String
  name) {
   //...
  }
}
```

The respective call in IVML would then look like

```
vilTemplateProcessor ("properties.vtl", config, target,
  name="myName");
```

Please note that this works with arbitrary types and that VIL type conversion applies. Further, since version 0.98, VIL and VTL support also named parameters on rules and templates that allow using default values without re-defining multiple versions of a basic implementation just with different signatures.

4.2.2 Appending or Prepending

While in some situations the complete creation of an artifact is required, in others it is sufficient to append or prepend information to the contents of an artifact.

The Imperative Style

Basically, we may obtain the contents of the artifact in terms of its textual representation and use the provided operations, e.g.,

```
def main(Configuration config, FileArtifact target) {
  Text targetText = target.getText();
  targetText.append("\n");
  targetText.append("Information ${config.name()}\n");
}
```

The modifications to the text representation will be synched into the artifact as soon as the target variable is reclaimed by the runtime environment, i.e., at the end of the subtemplate. The advantage of this approach is that it works in the same way in the VIL script so that a template may be superfluous. However, stating the required operation in each line and explicitly caring for the line ends is tedious.

Mixing contents

As an alternative, we can simply use the targetText variable within a content statement:

```
def main(Configuration config, FileArtifact target) {
  Text targetText = target.getText();
  '${targetText.text()}

   Information ${config.name()}'
}
```

Please note that a text representation is not automatically converted into a String in order to emphasize that the resulting String is disconnected from the underlying artifact while operations on the text representation will affect the artifact.

4.2.3 To format or not to format

Basically, VTL allows for explicit control of the formatting. This is intended if in certain contexts, e.g., industrial build settings, artifacts (after manipulation or generation) must be formatted exactly in a certain way or even pre-existing parts shall not be touched at all. This shall be supported by artifact implementations through individual formatting settings, e.g., the XML file artifact allows taking control over the auto-formatting applied while writing. However, in certain situations this may be intended or not. Here, VTL can be used in different ways:

- Rely on **auto-formatting** and do not care for the detailed formatting in your template (partial xTend style). Pass into a VTL script a certain artifact type for which a specific artifact implementation is known and let the artifact implementation do the formatting based on your formatting settings. Auto-formatting will be executed at the end of the VTL script when the whole content is set as text representation on the respective target artifact.
- Use an **implicit temporary artifact** to avoid formatting. Just create a `new FileArtifact()`, which is by default temporary and let VTL produce your content into. As temporary files are (by default) not associated with a certain artifact type, no formatting happens. Do not forget to move/rename the temporary artifact (auto-formatting does only apply while storing) at the end of your VTL script or better in the calling VIL script (process flow), as otherwise VIL will remove it automatically during cleanup after execution.
- Use an **explicit temporary artifact** to avoid formatting. Pass into your VTL file a path to a file type that is not associated with a certain artifact type and rename / move it afterwards.

4.3 All VIL languages

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

4.3.1 Rely on Automatic Conversions

Retrieving values from a decision variable may become lengthy due to the explicit type access to the actual variable.

```
Boolean value = config.byName(Variability).booleanValue();
```

As a shorter alternative, you may rely on automated conversions, i.e.,

```
Boolean value = config.byName(Variability);
```

4.3.2 Use Dynamic Dispatch

Some datatypes in IVML may be refined, e.g.,

```
compound Element {
// some variables
}
compound RefinedElement refines Element {
}
```

At a glance, processing these datatypes according to their types may lead (in VTL) to expressions such as

```
DecisionVariable elt = …
if (elt.type() == "Element") {
// … process basic elements
} else if (elt.type() == "RefinedElement") {
// … process refined elements
}
```

Actually, using alternatives to make this distinction is similar to object-oriented programming neglecting polymorphism, i.e., using if-cascades with type equality or `instanceof` expressions instead of overridden methods. In VIL and VTL, dynamic dispatch (a more generic form of polymorphism) can be used to achieve an elegant yet extensible specification. While dynamic dispatch is available for all static VIL/VTL types, in particular the artifact types, the IVML types come into play when using the `@advice` annotation (see Section 3.3.11 for details). Assuming that our specification is tagged by an appropriate `@advice` annotation, we can rewrite the specification above (here depicted for VTL but rules in VIL can be stated similary) to

```
def processElement(Element elt) {
// … process basic elements
}
def processElement(RefinedElement elt) {
// … process refined elements
}
```

and replacing the if-cascade by

```
processElement(elt);
```

Moreover, rules or sub-templates in refined scripts (VIL) or templates (VTL) may extend the processing (by adding new rules or sub-templates) or even override existing ones, respectively.

4.3.3 For-loop

Actually, VIL (in terms of `map`) and VTL (in terms of `for`) support iterations only over collections. For iterating over integers, just create a sequence of integers (`createIntegerSequence`) and iterate over that. Please note that this approach currently does not support arbitrary large integer ranges as noted in Section 3.4.5.3. We illustrate this in terms of a VIL fragment below.

```
Integer sum = 0;
sequenceOf(Integer) nums = createIntegerSequence(0, 10);
map(Integer i:nums) {
    sum = sum + i;
};
```

4.3.4 Create XML File / XML elements / XML attributes

XML files can easily be created by defining an `XMLFileArtifact` akin to an ordinary `FileArtifact`. However, initially an `XMLFileArtifact` is empty (except for the xml processing instruction), as no root element is defined. Thus, creating a structurally valid XML file looks like

```
XmlFileArtifact xmlFile = "$target/xml2File.xml";
XmlElement root = xmlFile.createRootElement("MyRoot");
```

Please note that the root element is created on file level, while for putting further elements or attributes into an XML file, the constructor can be used, such as

```
XmlElement hello = new XmlElement(root, "Hello");
new XmlAttribute(hello, "attrib", "value");
```

4.3.5 Overriding / Reinstantiating an XML File

The `XMLFileArtifact` is primarily intended for changing an existing XML file or, as described in Section 4.3.4, for creating new XML files. If an XML file shall be reinstantiated by rewriting a new XML file, you can either

- Delete all nodes in the root element and change the root element accordingy as deleting the root element is not possible due to technical restrictions. Please note that calling `createRootElement` a second time if a valid root exists does not lead to overwriting the root element.
- Delete the existing XML file first and then recreate the root element as needed as shown below.

```
XmlFileArtifact xmlFile = "$target/xml2File.xml";
xmlFile.delete();
XmlElement root = xmlFile.createRootElement("MyRoot");
```

4.3.6 Print some debugging information

Sometimes it is convenient while developing an instantiation to print out some information, e.g., about variable assignments. Therefore, VIL provides the global `println` operation, which takes any variable value and prints it to the console output. This may look like

```
Integer i = 1;
Integer j = 2;
println("$i $j");
```

4.3.7 Focus execution traces

VIL and VTL execution traces may become rather long and distract from understanding execution problems. Instead of running the full instantiation with tracing (typically the default depending on the utilized tool integration), you may disable this via

```
enableTracing(false);
```

completely or for selected parts of the instantiation.

5 Implementation Status

The development and realization of VIL and VTL related tools is still in progress. In this section, we summarize the current status.

Missing / incomplete functionality

- Collection of affected artifacts in VIL e.g., through `map` may be incomplete.
- VTL indentation formatting sometimes fails, in particular after we introduced composite expressions representing content statements with variables or expressions to be extrapolated. Here, VIL/VTL extensions for programming languages may help.

The implementation of the VIL/VTL languages utilizes Eclipse xText for generating parsers and editor support. Using VIL/VTL in Eclipse projects requires besides the EASy-producer nature also the xText nature and the xText builder, the latter in particular to update and display error and warning markers. However, the xText builder is rather resource intensive and so we disable its operations by default on all files and folders in `bin` and `target` of a project and focus its operations on the folders where VIL/VTL files are located (by default is the `EASy` folder, but may be adjusted e.g., to `src/main/easy` or `src/test/easy`). For now, we recommend adding the xText builder as last builder to `.project` to reduce its impact while enabling the marker functionality. So far, editor hooks did not help avoiding the xtText builder.

6 VIL Grammars

In this section we depict the actual grammar for the VIL languages. The grammar is given in terms of a simplified xText[29] grammar (close to ANTLR[30] or EBNF). Simplified means, that we omitted technical details used in xText to properly generate the underlying EMF model as well as trailing ";" (replaced by empty lines in order to support readability). Please note that some statement-terminating semicolons are optional in order to support various user groups each having individual background in programming languages.

6.1 VIL Grammar

```
ImplementationUnit:
  Import*
  Require*
  LanguageUnit*;

LanguageUnit:
  Advice* 'vilScript' Identifier '(' ParameterList? ')'
  (ScriptParentDecl)? '{'
    VersionStmt?
    LoadProperties*
    ScriptContents
  '}' ';'?

Require:
  'requireVTL' STRING versionSpec ';'

ScriptParentDecl:
  'extends' Identifier

LoadProperties:
  'load' 'properties' STRING ';'

ScriptContents:
  (VariableDeclaration | TypeDef | Compound | RuleDeclaration)*

RuleDeclaration: // Type is for future extension, ignored by now
  AnnotationDeclarations?
  (Type? RuleModifier? Identifier '(' (ParameterList)? ')' '=')?
  RuleConditions?
  RuleElementBlock ';'?

RuleConditions:
  (LogicalExpression)?  ':'
  (LogicalExpression (',' LogicalExpression)*)?
```

[29] http://www.eclipse.org/Xtext/

[30] http://www.antlr.org

```
RuleElementBlock:
  '{' RuleElement* '}'

RuleElement:
  VariableDeclaration
  | ExpressionStatement
  | While ';'?
  | For ';'?

RuleModifier:
  'protected'

ExpressionStatement:
    ((Identifier ('.' Identifier)? '=')? Expression ';')
    | Alternative ';'?

PrimaryExpression:
  ExpressionOrQualifiedExecution
  | UnqualifiedExecution
  | SuperExecution
  | SystemExecution
  | Map
  | Join
  | Instantiate
  | ConstructorExecution

LoopVariable:
  Type? Identifier

Map:
  'map' '(' LoopVariable (',' LoopVariable)* ('=', ':') Expression
')'
  RuleElementBlock ';'?

For:
  'for' '('LoopVariable (',' LoopVariable)* '=' | ':') Expression ')'
   RuleElementBlock

While:
  'while' '(' Expression ')'RuleElementBlock

Alternative:
    'if' '(' Expression ')' StatementOrBlock
    ('else' StatementOrBlock)?

StatementOrBlock:
    ExpressionStatement | RuleElementBlock

Join:
  'join' '(' JoinVariable ',' JoinVariable ')'
  ('with' '(' Expression ')')?

JoinVariable:
```

```
  'exclude'? Identifier ':' Expression
SystemExecution:
  'execute' Call SubCall*

Instantiate:
  'instantiate' (Identifier | STRING)
    '(' ArgumentList? ')' VersionSpec?
```

6.2 VIL Template Language Grammar

```
LanguageUnit:
  Import*
  Extension*
  Advice*
  IndentationHint?
  FormattingHint?
  'template' Identifier '(' ParameterList? ')'
  ('extends' Identifier)? '{'
    VersionStmt?
    (TypeDef
    | Compound
    | VariableDeclaration
    | VilDef)*
  '}'

HintedExpression:
  Expression ('|' (ID | '<'))

IndentationHint:
  '@indent' '(' IndentationHintPart (',' IndentationHintPart)* ')'

IndentationHintPart:
  Identifier '=' NUMBER

FormattingHint:
  '@format' '(' FormattingHintPart (',' FormattingHintPart)* ')'

FormattingHintPart:
    Identifier '=' (STRING | NUMBER)
;

VilDef:
  AnnotationDeclarations? 'protected'? 'def' Type? Identifier
  '(' ParameterList? ')' StmtBlock ';'?

StmtBlock:
    '{' Stmt* '}'

Stmt:
  VariableDeclaration
  | Alternative
  | Switch
  | Loop
```

```
  | While
  | ExpressionStatement
  | Content
  | Flush

Alternative:
  'if' '(' Expression ')' (Stmt | StmtBlock)
    (=> 'else' (Stmt | StmtBlock))?

Content:
  (STRING) ('|' Expression ';')?

Switch:
  'switch' '(' Expression ')' '{'
    (SwitchPart (',' SwitchPart)* (',' 'default' ':' Expression)?)
  '}'

SwitchPart:
  Expression ':' Expression

Loop:
  'for' '(' Type Identifier ':' Expression
    (',' PrimaryExpression)
      (',' PrimaryExpression)?
     ')'? (Stmt | StmtBlock)

While:
  'while' '(' Expression ')' (Stmt | StmtBlock)

Flush:
  'flush' ';'

Extension:
  'extension' JavaQualifiedName ';'

JavaQualifiedName:
    Identifier ('.' Identifier)*

SubCall:
  ('.' | '->') Call
    | '[' Expression ']'
    | '.(' (Type? Identifier '|')? '{' Stmt* '}'
```

6.3 Common Expression Language Grammar

Actually, parts of this common language are overridden and redefined by the two VIL language grammars.

```
LanguageUnit:
  Import*
  Advice*
  Identifier
  VersionStmt?
```

```
VariableDeclaration:
  'const'? Type Identifier ('=' Expression)? ';'

Compound:
    'abstract'? 'compound' Identifier ('refines' Identifier)? '{'
        VariableDeclaration*
    '}' ';'?

TypeDef:
  'typedef' Identifier Type '; '

Advice:
  '@advice' '(' QualifiedName ')' VersionSpec?

VersionSpec:
  'with' Expression

VersionedId:
  'version' VersionOperator VERSION

VersionOperator:
  '==' |'>' |'<' |'>=' |'<='

ParameterList:
  (Parameter (',' Parameter)*)

Parameter:
  Type Identifier ('=' Expression)?

VersionStmt:
  'version' VERSION ';'

Import:
  'import' Identifier VersionSpec? ';'

ExpressionStatement:
  (Identifier ('.' Identifier)? '=')? Expression ';'
Expression:
  LogicalExpression | ContainerInitializer

LogicalExpression:
  EqualityExpression  LogicalExpressionPart*

LogicalExpressionPart:
  LogicalOperator EqualityExpression

LogicalOperator:
  'and' | 'or' | 'xor' |'implies' | 'iff'

EqualityExpression:
  RelationalExpression EqualityExpressionPart?

EqualityExpressionPart:
```

```
  EqualityOperator RelationalExpression

EqualityOperator:
  '==' | '<>' | '!='

RelationalExpression:
  AdditiveExpression
  (RelationalExpressionPart RelationalExpressionPart?)?

RelationalExpressionPart:
  RelationalOperator AdditiveExpression

RelationalOperator:
  '>' | '<' | '>=' | '<='

AdditiveExpression:
  MultiplicativeExpression AdditiveExpressionPart*

AdditiveExpressionPart:
  AdditiveOperator MultiplicativeExpression

AdditiveOperator:
  '+' | '-'

MultiplicativeExpression:
  UnaryExpression MultiplicativeExpressionPart?

MultiplicativeExpressionPart:
  MultiplicativeOperator UnaryExpression

MultiplicativeOperator:
  '*' | '/'

UnaryExpression:
    UnaryOperator? PostfixExpression

UnaryOperator:
  'not' | '!' | '-'

PostfixExpression:  // for extension
  PrimaryExpression

PrimaryExpression:
  ExpressionOrQualifiedExecution
  | UnqualifiedExecution
  | SuperExecution
  | ConstructorExecution
ExpressionOrQualifiedExecution:
  (Constant | '(' Expression ')') SubCall*

UnqualifiedExecution:
  Call SubCall*
```

```
SuperExecution:
  'super' '.' Call SubCall*

ConstructorExecution:
  'new' Type '(' ArgumentList? ')' SubCall*

SubCall:
  ('.' | '->') Call | '[' Expression ']'

Declarator:
    Declaration (';' Declaration)* '|'

Declaration:
    Type? DeclarationUnit (',' DeclarationUnit)*

DeclarationUnit:
    Identifier ('=' Expression)?

AnnotationDeclarations:
    {AnnotationDeclarations}
    ('@' Identifier)*
Call:
  QualifiedPrefix '(' decl=Declarator? param=ArgumentList? ')'

ArgumentList:
  NamedArgument (',' NamedArgument)*

NamedArgument:
  (Identifier '=')? Expression

QualifiedPrefix:
  Identifier ('::' Identifier)*

QualifiedName:
  QualifiedPrefix ('.' Identifier)*

Constant:
  NumValue | STRING | QualifiedName | ('true' | 'false') | null
    | VERSION

NumValue :
  NUMBER

Identifier:
  ID | VERSION | EXPONENT | 'version'

Type:
  QualifiedPrefix
  | ('setOf' TypeParameters)
  | ('sequenceOf' TypeParameters)
  | ('mapOf' TypeParameters)
  | ('callOf' Type? TypeParameters)
```

```
TypeParameters:
  '(' Type (',' Type)* ')'

ContainerInitializer:
  '{' (ContainerInitializerExpression
    (',' ContainerInitializerExpression)*)? '}'

ContainerInitializerExpression:
    LogicalExpression |ContainerInitializer

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

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

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

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

terminal STRING :
  '"'
    ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|'"') )*
  '"' | "'"
    ( '\\' ('b'|'t'|'n'|'f'|'r'|'u'|'"'|"'"|'\\') | !('\\'|"'") )*
  "'"

terminal ML_COMMENT:
  '/*' -> '*/'

terminal SL_COMMENT:
  '//' !('\n'|'\r')* ('\r'? '\n')?

terminal WS:
  (' '|'\t'|'\r'|'\n')+

terminal ANY_OTHER:
```

References

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

[2] Eclipse Foundation. Xtend - Modernize Java, 2013. Online available at: http://www.eclipse.org/xtend.

[3] INDENICA Consortium. Deliverable D2.1: Open Variability Modelling Approach for Service Ecosystems. Technical report, 2011. Available online at http://sse.uni-hildesheim.de/indenica

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

[5] INDENICA Consortium. Deliverable D2.2.2: Variability Implementation Techniques for Platforms and Services (final). Technical report, 2013. Available online at http://sse.uni-hildesheim.de/indenica

[6] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: http://www.omg.org/docs/formal/06-05-01.pdf.

[7] 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].

[8] Richard M. Stallmann, Roland McGrath, and Paul D. Smith. GNU Make - A Program for Directing Recompilation - GNU make Version 3.82, 2010. Online available at: http://www.gnu.org/software/make/manual/make.pdf.

[9] The Apache Software Foundation. Apache Ant 1.8.2 Manual, 2013. Online available at: http://ant.apache.org/manual/index.html.

[10] S. Hallsteinsen and M. Hinchey and S. Park and K. Schmid, Dynamic Software Product Lines, IEEE Computer 41 (4), pages 93-95, 2008

[11] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, 2012.

[12] QualiMaster Consortium, Deliverable D4.1, Quality-aware Pipeline Modeling, Technical report, 2014, Available online at http://qualimaster.eu

[13] S.-W. Cheng, D. Garlan, B. Schmerl. Stitch: A Language for Architecture-based Self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, 2012.

[14] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, S. Kounev. Modeling Run-time Adaptation at the System Architecture Level in Dynamic Service-oriented Environments. *Serv. Oriented Comput. Appl.*, 8(1):73–89, March 2014.

[15] OMG Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.3, formal/2016-06-03

[16] H. Eichelberger, K. Schmid, EASy Variability Instantiation Language: Default Extensions, http://projects.sse.uni-hildesheim.de/easy/docs/vil_extensions.pdf [validated: June 2017].