IIP-Ecosphere

Integrated Variability Modeling Language: Language Specification

***Preview* version of 02. April 2025**

H. Eichelberger, S. El-Sharkawy, C. Kröher, K. Schmid

Software Systems Engineering (SSE)

University of Hildesheim

31141 Hildesheim

Germany

Abstract

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

In this document, we specify the concepts of the Integrated Variability Modelling Language (IVML) to describe customization and configuration options (in software ecosystems).

Version

1.0	15. February 2012	first version derived from D2.1
1.01	29. February 2012	“v” as prefix in version number (technical reasons)
1.02	15. June 2012	all parenthesis follow after all “with” keywords, accessing enum literals, camelcase for refTo and refBy
1.03	17. July 2012	DSL syntax corrected
1.04	19. July 2012	Constraint syntax, operation signatures and semantics, grammar section (prepared), technical section, import conventions, clarifications in using constraints
1.05	22. July 2012	grammar revised and added to document
1.06	06. August 2012	typeSelect, typeReject, side effects, undefined values
1.07	10. August 2012	examples testcased, enum access corrected in examples, container type definition adjusted (syntax overlap with variable declaration)
1.08	16. August 2012	Details for identifiers added, technical section deleted (see IVML user guide)
1.09	28. November 2012	Operator precedence in grammar corrected, constraint type added
1.10	12. December 2012	Mass assignment of annotation values
1.11	8. January 2013	Assignment operator clarifications (‘=’ vs. ‘==’)
1.12	17. January 2013	Eval clarified
1.13	11. February 2013	Compound / container initializers and annotations clarified
1.14	12. February 2013	Grammar cleanup, decision variable naming corrected
1.15	14. February 2013	Further clarifications on ‘=’
1.16	10. July 2013	Qualification clarification (reasoner dependent)
1.17	16. August 2013	Clarification on annotations, null values introduced
1.19	4. December 2013	Refined hierarchical import path.
1.20	21. April 2014	Implementation status
1.21	27. May 2014	Qualified names in interface exports.
1.22	03. July 2014 – 12. September 2014	Clarification of assignments in implies and enum ordinal operation. Type clarifications, hasDuplicates operation for Sequence. Alignment of container operation declaratory syntax. Dynamic dispatch for user-defined operations.
1.23	13. September 2014 – 2. January 2015	Freeze for projects, removal of DSL inclusion. Versioned import clarified. Version constraints for import / conflicts and Internal version type. Abstract compounds. Reserved keyword lists. Clarification of constraint variables and freeze. Value returning add function for container. `self` for constraints in compounds and fixes in operation list erroneously referring to self rather than operand.
1.24	12. January 2015	Implementation status for experimental SSE reasoner
1.25	24. March 2015	Clarification for multiple assignments in a project, change of freeze-but syntax. Importing user-defined operations, import sequence. Attribute renamed to annotation.
1.26	30. June 2015	Clarification on “Generics”, e.g. Set<Type> changed to setOf(Type) in the text, clarification on [] operator.

1.27	30. September 2015-15. March 2016	cleanup of multiple typedef constraints, flatten operations for set and sequence, allowing cyclic imports, clarifying collect. Eval in compounds, static user-defined operations. Min and max for ordered enums.
1.28	16. March 2016 - 18. July 2016	clarification for nested evals, const decision variables, extended apply, block expressions, convenience min/max container iterators
1.29	04. June 2018	Clarification how to use apply's return types, some OCL 2.4 alignment (additional / alias operations, asType, optional enum literal access via '::', optional access of usual container operations via '->' and '.', unqualified container iterators, implicit container iteration on single values, additional collection iterators such as selectByType, selectByKind, collectNested, sortedBy, closure, isAcyclic or iterate, toString for primitives, string operations and comparisons including locale), allInstances. Clarification of the evaluation sequence for isDefined, range comparisons, refining compound slots, compound multi-refinement. Shadowing for assignment blocks. Compound defaults vs. initializers.
1.30	01. April 2021 -	The copy function for all types, annotation and freeze shortcut with ".", clarifications for closure and product operations. Wildcard imports and experimental insert of user-defined functions to extend dynamic dispatch. Introducing operation annotations (so far without semantics). Immediate const freezing, relaxed initializer syntax, isDefined vs. forceDefined.

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 Bartu Dernek (content assist) and Adam Krafczyk for their contributions. We would also like to thank Stefan Krüger (TU Darmstadt) for his suggestion on min/max of ordered enums.

Table of Contents

Table of Figures

1 Introduction

This document specifies the Integrated Variability Modelling Language (IVML) in terms of a document containing the syntax and semantics of the language elements of the current version.

This document encompasses:

- The Variability Modeling Approach in terms of the
  - Core language,
  - Advanced concepts,
  - Constraint language
  - Built-in operations and types.
- The implementation status.
- The IVML grammar.

2 The Integrated Variability Modelling Approach

In this section, we will describe the concepts of the Integrated Variability Modelling Language (IVML). We distinguish between a core modelling language and an advanced modelling language that extends the core language. This distinction facilitates ease of use for the most standard issues in variability modelling, as it does not complicate the use of this language for users who do not need the more advanced features.

The basic concepts of the IVML relate to approaches like the Text-based Variability Language (TVL) [2], the Class Feature Relationships (Clafer) [1], the Compositional Variability Management framework (CVM) [7], etc.

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

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

- The syntax as well as the examples will be illustrated in `Courier New`.
- **`Keywords`** will be highlighted using bold font.
- *`Elements and expressions`* that will be substituted by concrete values, identifiers, etc. will be highlighted using italics font.
- Identifiers will be used to define names for modelling elements that allow the clear identification of these elements. We will define identifiers following the conventions typically used in programming languages. Identifiers may consist of any combination of letters and numbers, while the first character must not be a number. We recommend that the identifiers of new types start with a capital letter to easily distinguish them from variables.
- Expressions will be separated using semicolon "`;`".
- Different types of brackets will be used to indicate lists "`()`", sets "`{}`", etc. This is closely related to the Java programming language.
- We will indicate comments using "`//`" and "`/* ... */`" (cf. Java, C++).

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

- **Syntax:** this is the syntax of a concept. We will use this syntax to illustrate the valid definition of elements as well as their combination.

- **Description of syntax:** provides the description of the syntax and the associated semantics. We will describe each element, the semantics and their interaction with other elements in the model.
- **Example:** the concrete use of the abstract concepts is illustrated in a (simple) example.

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

In Section 2.2 we will describe the advanced concepts of the Integrated Variability Modelling Language. We will introduce extensions that are required to satisfy the specific requirements in particular drawn by in the FP7 INDENICA project like the support for service-ecosystems, for service technology and meta-variability.

2.1 Integrated Variability Modelling Core Language

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

2.1.1 Reserved keywords

The following keywords are reserved and must not be used as identifiers. Please note that this set of keywords is complemented by the keywords of the advanced modeling language concepts in Section 2.2.1 and the constraint language in Section 3.1.1.

- `abstract`
- `Boolean`
- `compound`
- `const`
- `Constraint`
- `enum`
- `false`
- `Integer`
- `project`
- `refine`
- `Real`

- refBy
- **refTo**
- **sequenceOf**
- **setOf**
- **String**
- **true**
- **typedef**
- **with**

2.1.2 Projects

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

Syntax:

```
project name {

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

}
```

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

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

Example:

```
project contentSharing {

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

}
```

2.1.3 Types

In a project (cf. Section 2.1.1), different kinds of core modelling elements may be used to both represent the variabilities and define a configuration space appropriately. We will express these kinds as formal types in IVML, thus defining a (strongly) typed language. We distinguish between basic types, enumerations,

container types, derived and restricted types and compound types. These types can be used to declare or define concrete decision variables. All decision variables can be unset using the null keyword, i.e., explicitly assigning no value to a variable.

2.1.3.1 Basic Types

IVML supports as basic types Boolean (**`Boolean`**), integer (**`Integer`**), real (**`Real`**) and string (**`String`**) with their usual meaning. The names of the basic types are aligned to OCL [4]. These types support the definition of basic variabilities, e.g. the `Boolean` type may be used for modelling optional variabilities. In addition, types like `Integer` or `Real` provide a basis for defining advanced variabilities, e.g. using an `Integer` to define a quantitative property. IVML provides the basic type `Constraint` which allows declaring constraints themselves as variables.

2.1.3.2 Enumerations

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

Syntax:

`enum` *Name*$_1$ {*value*$_1$, ..., *value*$_n$};

`enum` *Name*$_2$ {*value*$_1$=n_1, ..., *value*$_n$=n_n};

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

- The keyword **`enum`** defines the identifier *Name* as a new enumeration.
- *Name* is an identifier and defines the name of the new type.
- The identifiers surrounded by curly brackets are the concrete elements of the enumeration. A specific element of an enumeration can be accessed using the "."-notation, e.g. *Name*$_1$.*value*$_1$.
- Specifying concrete numeric values for elements of an enumeration (*value*$_i$=n_i) turns the enumeration into an ordered enumeration. This enables relations like greater than (>) or less than (<) and operations like next (**`next`**) or previous (**`previous`**) on the values to be used.

Example:

```
enum Colors {green, yellow, black, white};

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

2.1.3.3 Container Types

The IVML provides two container types, sequences and sets. Sequences can contain an arbitrary number of elements of a given content type (including duplicates), while sets are similar to sequences, but do not support duplicate elements. These types

can be used to describe a number of possible options out of which several can be selected at the same time. Elements in a container (both sequences and sets) can be accessed by their position in the container using an index (`[index]`). The allowed number of elements in a container, i.e., its cardinality, can be restricted by constraints.

The IVML supports a set of operations specific for container types, e.g. adding or appending elements to a container, deleting elements of a container, selecting specific elements, etc. We will introduce the full set of operations in Section 3.6.

Syntax:

```
// Declaration of a new sequence and a new set.

sequenceOf(Type) variableName1;

setOf(Type) variableName2;


/* Access to elements of a sequence. Sets do not have
index-based access. We will discuss variables in Section
2.1.4. */

variableName1[index] = value;
```

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

- The **`sequenceOf`** and **`setOf`** keywords refer to a container of the respective type followed by the *`Type`* of the elements contained in brackets.
- The identifiers *`variableName`*$_1$ and *`variableName`*$_2$ are the names of the new containers.
- Accessing a specific element of a sequence container type (variable) requires the specification of an index (`[index]`). An index is either “0” or a positive integer value specifying the position of an element in a container. Accessing a specific position is only a valid operation, if this position has previously been set by different means like the **`add`** function (the set of operations is introduced in Section 3.6).

Example:

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

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

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

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

2.1.3.4 Type Derivation and Restriction

The IVML allows the derivation of new types based on existing types. This supports extensibility and adaptability as users may define their own types based on basic types, enumerations or container types as well as on previously derived types. The derivation may also include restrictions to the existing type, e.g. to restrict the possible values of the new type to a subset of the values of the existing type. The optional restrictions are defined by a constraint in OCL style (we will discuss constraints in detail in Section 3).

Syntax:

```
typedef Name1 Type;

typedef Name2 Type with (constraint);
```

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

- The **typedef** keyword indicates the derivation of a new type based on an existing type.
- The identifiers *Name1* and *Name2* are the names of the new types.
- The identifier *Type* denotes the basic type from which the new type (*Name1* or *Name2*) will be derived.
- The optional keyword **with** defines a constraint (cf. Section 3), surrounded by brackets, which must hold for *Name2* (*Name2* can be used as identifier in constraint), e.g., if deriving *Name2* from **String** the constraints may define regular expressions based on *Name2*.

Example:

```
/* Definition of a type "AllowedBitrates" which is a set
of Integers, i.e. a kind of alias for a complex type
definition. */

typedef AllowedBitrates setOf(Integer);


/* A new modelling type of the basic type integer that is
restricted to assume values between "128" and "256". */

typedef Bitrate Integer with (Bitrate >= 128 and
  Bitrate <= 256);
```

2.1.3.5 Compounds

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

Syntax:

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

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

- The optional keyword **`abstract`** indicates that this specific compound cannot be instantiated. Anyway, it can be refined, e.g., serve as a root of compound types.
- The **`compound`** keyword indicates the definition of a new compound type.
- The identifier *Name* defines the name of the new compound type.
- The set of slot elements surrounded by curly brackets defines the types of the compound type. Slots are separated by semicolon.
- Optional constraints defined on the slots of the compound that must hold for all compound instances.

Example:

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

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

2.1.4 Decision Variables

The types introduced in Section 2.1.3 can be used to declare (decision) variables representing a concrete variability. A decision variable is an element of a project (configuration space) that accepts any value of its type. Constraints may further restrict the possible values by removing certain combinations of values from the

allowed configuration space. The value given to a decision variable defines the variant of the represented variability.

Either a decision variable may be declared with or without a default value (this is an optional parameter). Decision variables with a default value can be further configured by overwriting their (default) value at a later point in time. Overwriting the default value is not necessary. However, due to the declarative nature of IVML, i.e., the evaluation sequence of constraints is not coupled to the specification sequence for most concepts, variables can be modified only once per project scope and further re-assignments must not occur in the same project scope (see also Section 3.1.4). Decision variables can be constant, i.e., assigned once by a default value, stay with that value and must remain unchanged. Default values become effective when a variable is being assigned. This is in particular relevant for compound types, where an instance of the type is created upon first assignment of a slot or of a complex compound value.

Syntax:

```
// Declaration of a decision variable.

Type name1;

/* Declaration of a decision variable with a default
value. The "valueAssignment1"-expression will be described
in detail below. */

Type name2 = valueAssignment1;

const Type name3 = valueAssignment2;
```

Description of Syntax: The basic declaration of a new decision variable (excluding the declaration of an optional default value) consists of the optional keyword '`const`', the desired type (one of the basic types, an enumeration, a container type, a derived or a restricted type, or a compound type) followed by an identifier (*name*$_1$) that states the name of the variable.

Optionally, except for constants, a default value can be assigned to a decision variable appending "=" followed by a "*valueAssignment*"-expression after the name (*name*$_2$) of the decision variable. The form of the "*valueAssignment*"-expression depends on the specific type of the declared decision variable:

- Basic types and Enumerations: An expression that yields a value of the corresponding type and can be actually evaluated to a constant or from variables for which the values are known.
- Container types: either an expression of the type of the container, which can be statically evaluated, or a set of values separated by commas in curly brackets after the name of the decision variable. Expressions may be used but must be stated in parenthesis due to technical reasons. The allowed

values within the curly brackets are determined based on the base type of the container.

- Compounds: either an expression of the type of the compound, which can be statically evaluated, or a set of individual assignments, given in curly brackets. Each assignment explicitly gives the field in the compound that the assignment is made to, followed by a "=" and an expression of the corresponding element type. Again this expression needs to be statically evaluated.
- Derived type: the assignment follows the rules of the base type.

`const` variables behave like constants and are, thus, subject to immediate freezing (cf. Section 2.2.5.2) at value assignment.

Example:

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

Integer bitrate = 128;


/* Declaration of a new variable of type enumeration with
a default value (cf. Section 2.1.3.2). */

Colors backgroundColor = Colors::black;


/* Declaration of a new variable of type container
(sequence) with default values (cf. Section 2.1.3.3). */

sequenceOf(ContentType) baseContent =
   {ContentType::text, ContentType::audio};


/* Declaration of a new variable of type compound with
default values (cf. Section 2.1.3.5). */

Content complexContent = {name = "Text",
  bitrate = 128};
```

Compound values are written in terms of compound initializers, i.e., as attribute-value assignment list in curly brackets separated by commas. To ease writing configurations, the attribute-value assignment list may end with a (purely syntactical) single comma without following attribute-value pair.

2.1.5 Configurations

The IVML does not differentiate between a configuration space and specific (product) configurations. Instead, a project can simultaneously describe or extend a configuration space and define a configuration. However, typically a project will provide a configuration space, while a different project may extend it, while providing configurations information for the initially specified configuration space. The set of decision variables and constraints of a project represent the set of all possible configurations. In addition, default values of decision variables as described in Section 2.1.4 define basic configurations and, thus, do not need to be further configured, but can be overwritten later as well. In addition, some values of decision variables can be derived using constraints. Any configuration, independent of where the values come from, must comply with the relevant constraints.

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

Example:

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

project contentSharing {

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

  typedef Bitrate Integer with (Bitrate >= 128 and
    Bitrate <= 256);

  ContentType content;

  Bitrate contentBitrate = 128;

  contentBitrate == 128 implies
    content == ContentType::text;

}
```

2.2 Advanced Concepts of the Integrated Variability Modelling Language

This section describes advanced concepts of the IVML. We will describe how to assign additional annotations to modelling elements. This allows describing certain modelling elements in more detail, e.g. assigning meta-variability information. We then augment the compound types introduced in Section 2.1.3.5 by extension and referencing concepts. Extension concepts will also be introduced for projects (cf. Section 2.1.1), which cover modularization aspects as well as facilitating project

composition. Finally, we will describe advanced configuration concepts including partial configurations as well as “freezing” configurations.

2.2.1 Reserved Keywords

The following keywords are reserved and must not be used as identifiers. Please note that this set of keywords is complemented by the keywords of the basic modeling language concepts in Section 2.1.1 and the constraint language in Section 3.1.1.

- **`assign`**
- **`annotate`**
- **`but`**
- **`conflicts`**
- **`eval`**
- **`export`**
- **`freeze`**
- **`import`**
- **`insert`**
- **`interface`**
- **`static`**
- **`to`**
- **`version`**

2.2.2 Annotations

In the IVML modelling elements can be annotated by further (orthogonal) configuration capabilities, e.g. to express meta-variability such as binding times. An annotation in IVML is basically a decision variable that is attached to another modelling element describing this element in more detail. Thus, an annotation may also have a default value and may be restricted by constraints (cf. Section 3). The impact of an annotation depends on the element it is attached to. In the IVML the following modelling elements can be annotated:

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

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

Syntax:

```
annotate Type name1 to name2;

annotate Type name3 = value to name4;
```

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

- The **annotate** keyword[1] indicates the definition of a new annotation.
- The expressions *Type name1* and *Type name3* correspond to the definition of a decision variable described in Section 2.1.4 while *name1* and *name3* are the identifiers of the new annotations[2].
- The **to** keyword indicates the attachment of the new annotation on the left side to the element (*name4*) denoted on the right side. Multiple names may be given separated by commas
- *name4* may be one of the elements described above to which the annotation is attached.
- Optionally, a default value (*value*) can be assigned to the annotation by appending a value expression after *name3*.

Example:

```
project contentSharing {

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

  // Attaching an annotation to the entire project.

  annotate BindingTimes binding = BindingTimes::compile
    to .;

}
```

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

```
compound Content {

  String name;

  Integer bitrate;

}

Content content;
```

[1] The keyword **attribute** is deprecated, but still recognized by the implementation. However, it may be removed completely in one of the future versions.

[2] Due to technical reasons, currently annotations must not start with 'v' or 'e'.

```
annotate BindingTimes binding = BindingTimes.compile
    to content;

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

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

Example:

```
project contentSharing {

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

  // Attaching an annotation to the entire project.

  annotate BindingTimes binding = BindingTimes::compile
    to contentSharing;

  enum Colors {black, white};

  Bitrate contentBitrate = 128;

  contentBitrate.binding = BindingTimes.configuration;

  Colors backgroundColor = Colors::black;

  backgroundColor.binding = BindingTimes::configuration;

  // go on with several variables and different binding

  // times

}
```

IVML provides the assign construct as syntactic sugar to simplify the mass-assignment of values to annotations and to visually group the model elements with same (initial) annotation assignment. However, the variables "declared" in the assign block actually are part of the containing element, in the example below the project `contentSharing`. Constraints for the "declared" elements can be stated within the assignment block, but actually belong to the enclosing block. An assign block can also be used within a compound, it may even be nested in other assign blocks if needed or multiple annotations may be given in comma-separated fashion in the parenthesis of an assign block. If multiple assignments defined by nested blocks refer to the same annotation, for convenience only the innermost one is valid, i.e., the outer

assignments are shadowed. Technically, an assign block is syntactic sugar and translated into individual assignment constraints (‘=’) according to the parenthesis of the respective assign blocks. Shadowed assignments shall not be translated, i.e., shall not cause re-assignments within the same scope.

Example:

```
project contentSharing {

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

  // Attaching an annotation to the entire project.

  annotate BindingTimes binding = BindingTimes::compile
    to contentSharing;

  enum Colors {black, white};

  assign (binding = BindingTimes::configuration) to {

    Bitrate contentBitrate = 128;

    Colors backgroundColor = Colors::black;

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

  }

}
```

2.2.3 Advanced Compound Modelling

In Section 2.1.3.5 we introduced the compound types to group multiple types into a single named unit. In this section, we will extend the modelling of compound types by refinement and referencing concepts. Refinement allows extending existing compound types by additional elements, yielding a new (extended) compound type. Referencing enables the definition of references to other elements like other compounds.

2.2.3.1 Extending Compounds

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

Syntax:

```
compound Name1 refines Name2 {

  // Define additional elements.
```

```
}
```

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

- The **compound** keyword indicates the definition of a new compound type.
- The identifier *Name₁* defines the name of the new compound type.
- The **refines** keyword indicates that the new compound type (*Name₁*) is an extension of a previously defined compound type (*Name₂*). Multiple refining compounds can be given as comma-separated list.
- The set of elements surrounded by curly brackets defines the additional elements, e.g., slots, constraints that make up the extensions to the inherited elements of compound *Name₂*. New slots may override/shadow already defined slots in the refined compound(s) through specifying a slot with the same name only if the overriding compound is type compatible, i.e., the new slot is of the same or of a sub-type of the directly preceding slot in the compound refinement hierarchy. This restriction is required to ensure substitutability among compounds of the same hierarchy. Currently, constraints for shadowed slots do not apply to the new slots.

Example:

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

compound Content {

  String name;

  Integer bitrate;

}


/* A new compound type that refines the previous compound
type. "ExternalContent" will subsume all elements of
"Content" and all additional elements defined below. */
compound ExternalContent refines Content {
  String contentPath;
  String accessPassword;
}
```

2.2.3.2 Referencing Elements

The IVML supports referencing of (other) elements, for example, other compounds within a compound type. A reference allows the definition of individual configurations of an (external) element for the referencing element without including the external element as part of the referencing element explicitly. This is

indicated by the **refTo** keyword used for the definition of a reference and the **refBy** keyword that indicates the configuration of a referenced element.

Syntax:

```
project name1 {
  compound Name2 {
    Type name3;
    ...
  }

  // Declaration of a new reference.
  refTo(Name2) Name4;

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

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

- The **refTo** keyword indicates the definition of a new reference.
- *Name2* defines the referenced element (type).
- *Name4* is an identifier and defines the name of the new reference.
- The **refBy** keyword indicates the configuration of a reference (the configuration of the referenced element respectively).
- *Name4* is an identifier that defines the reference to be configured.

Example:

```
/* A compound type for the configuration of different web
containers being responsible for serving web content. */
compound Container {
  String name;
  ...
}
```

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

compound Content {

  String name;

  Integer bitrate;


  // Declaration of a reference to the Container compound.

  refTo(Container) myContainer;


  // Configuration of the above reference.

  refBy(myContainer).name = "ContentContainer";

}
```

2.2.4 Advanced Project Modelling

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

2.2.4.1 Project Versioning

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

Syntax:

```
project name{

  // Definition of a version for this project
```

```
    version vNumber.Number;
    ...
}
```

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

- The **version** keyword indicates the definition of a new version for the project *name*.
- *vNumber.Number* defines the actual version of the project (here only two parts prefixed by a "v"). At least one number must be given and no restriction holds on the amount of sub-version numbers.

Example:

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

2.2.4.2 Project Composition

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

- **import**: this keyword[3] indicates the use of a project. An imported project is evaluated before import, thus an import acts as an implicit eval. Multiple import statements are processed in the given sequence, i.e., while resolving imported model elements, the first matching along the import takes precedence. Multiple, but inconsistent definitions shall lead to an ambiguity error. Imported elements are either available through their simple name (local package takes precedence over first import declaring the name) or qualified names using '`::`' as namespace separator. The name may end with a * indicating a wildcard import of projects with the given name prefix. Except for self-imports, cyclic imports shall be processed and imported model

[3] Since September 2022, there is an experimental import variant indicated by `insert` keyword (to be used instead of `import`). While `import` makes the referenced language concepts available, `insert` virtually inserts user-defined functions into the project(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.

elements shall be resolved wherever possible unless elements cause themselves resolution cycles. Self-imports and model elements with resolution cycles shall cause error messages. This keyword allows using certain elements of a project by reference. If a project contains explicit interfaces (see below), the specific interface, which is used, must be given.
However, multiple projects with identical names and versions may exist in a file system[4], in particular in hierarchical product lines. Thus, project imports are determined according to the following **hierarchical import convention**, i.e. starting at the (file) location of the importing project (giving precedence to imports in the same file) the following locations are considered in the given sequence: The same directory, then contained directories (closest directories are preferred) and finally containing directories (also here closest directories are preferred). In addition, sibling folders of the folder containing the importing model and predecessor projects are considered[5]. Similar to Java class paths, additional model paths[6] may be considered in addition to the immediate file hierarchy.

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

The keywords **`import`** and **`conflicts`**, introduced above, can be combined with expressions using the **`with`** keyword[7], e.g., limiting the version information of a project (see Section 2.2.4.1). The version of the import can be referred by the keyword version or using the name of the import project followed by ".version". The internal version type (cf. Section 3.2.3) defines relational operators such as '<', '>', '<=', '>=','==', '<>' or '!='. More complex expressions can, e.g., be composed using Boolean operations. In case of multiple matching versions, the model with the highest version number is selected by default. Please note that constant version numbers start with "v" (cf. Section 2.2.4.1).

Syntax:

```
project name1 {
```

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

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

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

[7] Before version 1.23, IVML required parenthesis around a version restriction and supported a limited form of expressions. Since version 1.23, IVML supports complex expressions similar to constraints. Further versions will extend these capabilities.

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

    import name2;

    // Accessing elements of a project.

    name2::element;


    /* This introduces incompatibility of project name1 with
    project name3 of version complying with expression. */

    conflicts name3 with expression;

}
```

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

- The keyword **`import`** indicates that the entities, which are made available by the project or interface *name₂* will be available within the current project.
- For disambiguation the elements of *name₂* can be accessed using the "`::`"-notation to express qualified names. If there is no ambiguity, they can be used directly.
- The keyword **`conflicts`** indicates incompatibility of project *name₁* with project *name₃*.
- Optionally, version-expressions can be combined with the keywords **`import`** and **`conflicts`** using the **`with`** keyword. This defines specific versions of other projects to be imported into the current project or conflicting with the current project.
- A version expression includes the version-information of a project (cf. Section 2.2.4.1), in particular the relation operations defined by the internal version type (cf. Section 3.2.3) and version constants (starting with a "v" as defined in Section 2.2.4.1) and a version number or a version-information of another project. In addition, logical operators can be used to concatenate simple version-expressions to define ranges of versions.

Example:

```
project application{

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

```
  String name;
}

project targetPlatform {
  // This will define a new project for target platforms.
  version v1.5;
  String name;
}

project contentSharing {
  /* This will define a new project for a content-sharing
  project importing two sub-projects "application" and
  "targetPlatform". The latter sub-project must be of
  version "1.3" or higher. */
  import application;
  import targetPlatform
    with (targetPlatform.version >= v1.3);

  // Accessing the elements of the sub-projects.
  application::name = "myApp";
  targetPlatform::name = "myPlatform";
}
```

2.2.4.3 Project Interfaces

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

Interfaces in a project define all elements of a project, not part of the interface, as private and, thus, make them invisible to the outside. This is indicated by the **`interface`** keyword within a project. In order to access any element it must be declared as parameters of the interface. This can be done by exporting existing variables (using the **`export`** keyword) or by declaring new parameter variables. As a special characteristic of the IVML, it is also possible to define multiple interfaces for

the same project. This is different from other variability modelling languages like the CVL [6].

Importing a project (cf. Section 2.2.4.2) that includes interfaces allows the importing project to access only the parameters defined in the interface. All other elements of the project are not visible to the importing project.

Syntax:

```
project name1 {

  // Definition of a new interface.

  interface Name2 {

    /* Denotes the export of an existing decision variable
    of the project name1. */

    export name3;

    ...

  }

  /* Declaration of a (private) decision variable. This
  variable is exported by the interface Name2. */

  Type name3;

}
```

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

- The keyword **`interface`** indicates the definition of a new interface of the project *name*$_1$. Interfaces must occur at the beginning of a project before decision variable or type definitions.
- The keyword **`export`** indicates the export of the following decision variable *name*$_3$.

Example:

```
project application {

  // This will define an interface for this project.

  interface MyInterface {

      export name, appType;

  }
```

```
    // Declaration of (private) decision variables.
    String name;
    String appType;
    Integer bitrate;

    // Definition of a constraint.
  appType == "Video" implies bitrate == 256;}

  project contentSharing {
    /* This will import the interface "MyInterface" of
    project "application". */
    import application::MyInterface;

    /* Only the parameters of the interfaces are accessible.
    "application::bitrate" yields an error. As long as the
    variable names are unambiguous, the fully qualified must
    not be used. */
    name = "myApp";
    appType = "Video";
  }
```

2.2.5 Advanced Configuration

In Section 2.1.5, we introduced the configuration concept of the IVML. In this section, we will extend this concept to partial configuration. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring only parts of the project. The set of all configuration steps typically yield a full configuration of the entire project. We will further introduce the concept of persistent (parts of) configurations. We call this “freezing”. Freezing (parts of) configurations defines these parts to be persistent. Persistent parts cannot be changed anymore in further configuration steps. Finally, we will describe how (parts of) configurations can be evaluated independently from other parts of the configuration. This allows deriving additional configuration values based on existing configurations using the constraints and value propagation.

2.2.5.1 Partial Configurations

The IVML supports partial configurations. Partial configuration allows the configuration of a project in terms of multiple configuration steps, each configuring

only parts of the project. The set of all configuration steps typically yields a full configuration of the entire project. The configuration of a part of a project may also be reconfigured by the next configuration step (cf. the concept of default values, which we introduced in Section 2.1.4). For example, a service provider may define a (pre-) configuration of the provided service, while a service consumer may reconfigure his service to satisfy his specific needs.

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

Example:

```
project application {

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

  String name = "Application";

}


project targetPlatform {

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

  String name;

}


project contentSharing {

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

  import application;

  import targetPlatform;


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

  application::name = "myApp";
```

```
    targetPlatform::name = "myPlatform";
}
```

2.2.5.2 Freezing Configurations

In the previous section we described the concept of partial configuration. This included the possibility to re-configure existing (pre-) configurations. Although re-configuration is reasonable in some cases, e.g. to modify a given configuration to satisfy an individual need, at the end we desire a persistent configuration to define a specific product. This is particularly needed in the context of variability implementation techniques that remove parts that are not needed. For them, the freeze signals that some parts can be removed.

We introduce the concept of "freezing" configurations. This is indicated by the keyword **`freeze`** containing variable statements to freeze. Freezing configurations define the current (partial) configuration to be persistent. Similar to declaring annotations, the containing project can be stated as a dot (".”). Persistent configurations cannot be changed anymore in the course of the configuration. Constant variables are frozen immediately at assignment (cf. Section 2.1.4).

Excluding elements of a configuration from being frozen, e.g. freezing only some elements of imported projects or a compound type, the **`but`** keyword can be attached after a freeze-expression. The **`but`** keyword is then followed by a selector expression in the style of the container operations shown in Section 3.1.16, i.e., an iterator variable visiting all identifiers given in the freeze block and a Boolean expression using that iterator variable. By default, the iterator variable is of the internal type `FreezeVariable` see Section 3.2.4 for details and provides access to some variable information such as the name as well as the annotations of all elements defined within the freeze block (in case of type conflicts of annotations with the same name, the first annotation definition in sequence of the elements in the freeze block counts). The iterator variable is resolved locally, i.e., no other variable is visible. All elements matching that expression will not be frozen, however if the expression remains undefined during evaluation a freeze will happen anyway. Freezing always happens at the end of a scope evaluation, multiple freezes in a scope happen in unspecified sequence.

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

Syntax:

```
project name1 {
    // Definition of new compound type
    compound Name2 {
        Type name3;
        Type name4;
```

```
    }

    /* Declaration of a new decision variable of the above
    type */
    Name2 name6;

    /* Freezing the configuration of the decision variable
    except element name4. */
    name6.name3 = value1;
    freeze {
      name6;
    } but (name7|expression)
  }
```

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

- The keyword **`freeze`** indicates that all specified elements with their current values within the following curly brackets are persistent. Elements are given as statements, which may be variables, qualified variables or projects (the containing project can be stated as "."). For convenience, the ";" of the last statement may be omitted.
- Optionally, the keyword **`but`** indicates that some elements shall be excluded from being persistent through freezing. This is expressed in terms of an iterator variable (*name7*) of type FreezeVariable (see Section 3.2.4 for details and operations) and a Boolean selector *expression* using the freeze variable determining those elements that shall not be frozen.

Example:

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

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

    ContentType appContent;

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

    appContent.contentName = "Text";

    freeze {

      appContent;

    } but (f|f.name() == "bitrate")

}
```

Following the example from Section 2.2.2, freezing a project without runtime variables declared through a binding time annotation may looks like:

```
    freeze {

      contentSharing; // alternative: .;

    } but (v|v.binding == BindingTimes.runtime)
```

2.2.5.3 Partial Evaluation

Constraints in IVML do not imply a certain evaluation sequence. However, in some situations the domain engineer wants to indicate a certain priority, e.g., that some form of type dependent initialization of (compound slot) variables must always be done before evaluating other constraints in the same containing scope. Thus, IVML provides a concept for enforcing the evaluation of configurations. This is indicated by the keyword **`eval`** followed by a block either containing further evals or constraint statements. The explicit declaration of *nested* **`eval`** structures (always before the constraint expressions in an eval block) can be used to structure the definition of the evaluation and, thus, reduces the search-space during constraint-evaluation. By default, the top-level **`eval`** structure is the containing project, i.e., at the end of a project definition an implicit **`eval`** over all constraints defined in the project occurs. **`eval`** structures can be given in scopes where constraints can be defined, i.e., in projects or compounds. While nested **`eval`** structures enforce an inside-out evaluation, **`eval`** structures on the same nesting level do not imply a sequence of evaluation as this is the case for constraints in a project. An **`eval`** block enables this forced evaluation only once upon the first evaluation of the containing scope.

Currently, an **`eval`** block may only contain constraints, i.e., variables belong to the containing scope (project or compound) and no variables can be defined in an **`eval`** (this may change in future, then variables would be propagated from inside the **`eval`** the outside **`eval`** or project). Further, currently assign blocks cannot contain **`eval`** blocks, but also this may change in future.

Syntax:

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


eval {

  name1 = name2;

}
```

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

Example:

```
project application{

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

  compound ContentType {

    String contentName;

    Integer bitrate;

  }

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

  ContentType appContent;

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

  eval {

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

}
```

```
project targetPlatform {

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

  String name;

  Integer bitrate;

}


project contentSharing {

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

  import application;

  import targetPlatform;

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

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

}
```

3 Constraints in IVML

In this section we will describe syntax and semantics of the IVML constraint sublanguage. In Section 3.1 we will describe the constraint language and in Sections 3.2 to 3.7 the built-in operation which can be used within constraint expressions.

3.1 IVML constraint language

In this section we will define the syntax and the semantics of the IVML constraint language. The notation of constraints in IVML is inspired by OCL, a well-known language to express constraints on MOF/UML models. In more detail, the IVML constraint language is based on a systematic selection of OCL concepts driven by the concepts and types defined by IVML. In addition, due to the nature of IVML as a configuration modeling language, IVML adds additional operations to enable side-effects, i.e., to change and propagate values during model validation and reasoning. Therefore, most of the content in this section is taken from OCL [4] and adjusted to the notational conventions and the semantics of IVML.

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

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

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

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

- `(10 <= a and a <= 20) implies b == a;`
- If `a` is in the range `(10; 20)` this implies that `b` must have the same value as `a`. Alternatively, one can state the same constraint using a range comparison expression `(10 <= a <= 20) implies b == a;`
- `1 <= mySet.size() <= 100;`
  Cardinality restriction of mySet containing arbitrary decision variables.
- `mySet->forAll(x|x > 100);`
  All elements in `mySet` must be larger than `100`

Constraints may be used in two distinct ways in IVML:

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

Both, standalone constraints and embedded constraints may refer to individual variables as well as to types. In the first case, the constraint applies to the specific variable only. The second case occurs if a constraint is specified within a compound, i.e., it applies to the slots of all compound instances of that type. Further, this case may occur in a standalone constraint if the compound is referenced by its type name. During constraint evaluation, implicit “static” access to a compound slot or a type is basically unbound and needs constraint rewriting, i.e., the unbound variables are instantiated for all instances of that type. Thereby, compound slot accesses are grouped in order to reduce the number of introduced qualifiers, i.e., all accesses to the same type of compound are handled by the same instance. In case that other constraint semantics are intended, collecting all instances in the model and explicitly quantifying the expression is required.

In addition, IVML allows defining constraint variables, i.e., variables of type Constraint that can hold a variable. This allows to change and disable certain constraints at evaluation time, in particular in importing projects. When evaluating the constraints of a scope, the constraints in the constraint variables of the scope are evaluated as if they are defined as usual constraints.

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

3.1.1 Reserved Keywords

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

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

- **self**
- **then**
- **xor**

Please note that this list is complemented by the reserved keywords for the basic modeling concepts in Section 2.1.1 and the keywords for the advanced modeling concepts in Section 2.2.1.

3.1.2 Prefix operators

IVML defines two prefix operators, the unary

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

3.1.3 Infix operators

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

```
a + b
```

is conceptually equal to the expression:

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

that is, invoking the "+" operation on a (the *operand*) through the dot- access notation with b as the parameter to the operation. Note that there are limitations for applying this notation to the subtraction operator '–', as then unary and binary operation are ambiguous (similarly for Boolean negation). The infix operators defined for a type must have exactly one parameter. For the infix operators '<', '>', '<=', '>=', '<>', '**and**', '**or**', '**xor**', '**implies**', '**iff**' the return type must be Boolean.

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 IVML integer is a subclass of real. Thus, for each parameter of type real, you can use integer as the actual parameter. However, the return type will always be real. We will detail the operations on basic types in Section 3.2.4.

Further, please note that expressions on the left side of implications (implies) and two-sided implications (iff) must not be assignments ('=').

3.1.4 Equality and assignment operators (default logic)

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

Basically, a decision variable in IVML is considered as **undefined**, i.e., the variable does not have an effect on the instantiation. Default assignment and the equality

operator can influence whether a variable is defined or undefined. In particular, assigning the pre-defined value **`null`** overrides an already configured variable by a defined value indicating nothing, i.e., no explicit configuration. Please note that for instantiation all (relevant) decision variables must be frozen (cf. Section 2.2.5.2) and that also undefined decision variables can be frozen.

A **default value** can be assigned to a variable. Default values can be used to define a basic configuration (a kind of basic profile) which applies to all products in the product line. A default value can be defined as part of the variable declaration[8] (using the '=', cf. Section 2.1.4) or in terms of an individual default assignment using the '=' operator. Default values may be changed by partial configuration (cf. Section 2.2.5.1), i.e., on the import path of a (hierarchical) variability model the default value of certain decision variables may be modified in order to adjust the basic profile, e.g., to a certain application setting or domain. However, due to the mostly declarative nature of IVML, the value of a variable can be modified only once in a given model (assigned or changed). This restriction is required due to the fact that IVML does not provide support to define the sequence of evaluations (except for imports and eval blocks, cf. Section 2.2.5.3). As a default value is treated as a shortcut of an assignment, further assignments or changes of the default value must not be done in the same model, i.e., if a default value is assigned to variable *x* in project *P*, *x* may be changed in projects importing P unless frozen, but not directly in *P*. For deciding whether a re-assignment in the same scope happens, the declaration project scope of the constraint under evaluation is relevant.

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

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

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

Constraints may explicitly test the defined/undefined state of a variable using the operation `isDefined`. In combination with default assignment or equality for the same variable, the defined/undefined state of a variable may change during one evaluation and cause an implicit temporal effect on the result of `isDefined`. Therefore, an implementation shall evaluate `isDefined` after all relevant

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

assignments are done. While `isDefined` is undefined for variables that never have received any value (i.e., it can be used as a safe guard for disabling constraints on variables that may conveniently remain undefined), `forceDefined` fails if a variable is not configured in any form.

3.1.5 Precedence rules

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

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

'**`implies`**', '**`iff`**'Parentheses '`(`' and '`)`' can be used to change precedence.

3.1.6 Datatypes

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

- In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes.
- Enumerations literals can be specified as qualified names, either in IVML style using a dot to separate the enumeration from the literal, e.g., `package::MyEnum.literal` or in OCL style using the namespace separator `::`, e.g., `package::MyEnum::literal`. For a certain enumeration type the defined literals can be used with assignment ('='), equality ('==') or inequality ('!=', '<>') operators. In case of an ordered enumeration, also further operations such as relational operators ('<', '>', '<=', '>=') may be used. See Section 3.4.1 for operations on enumeration literals and Section 3.4.2 for operations on literals of ordered enumerations.
- Decision variable declarations defined within a compound can be accessed using the dot operator '`.`'. `self` refers to the value of a compound and can be used in (implicitly all-quantized) constraints within compounds.
- In addition to the string operations defined for OCL, we added two operations based on regular expressions, namely matches and substitutes.

3.1.7 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. Any is typically used for defining the built-in operations. The only value of Any is **`null`**, which overrides any configured value by a value representing nothing.
- Each type conforms to its (transitive) supertypes. Figure 1 depicts the IVML type hierarchy.
- Type conformance is transitive.
- The basic types do not comply with each other, i.e. they cannot be compared, except for Integer and Real (actually the type Integer is considered as a subclass of Real).
- Containers are parameterized types regarding the contained element type. Containers comply only if they are of the same container type and the type of the contained elements complies.
- The **`refines`** keyword induces a hierarchy of compounds where the subtypes are compliant to their parent types, i.e. the parent type may be replaced by each subtype.
- Derived types are compliant to their base type.
- MetaType is a specific type denoting types, e.g. to constrain types of elements within a container.

3.1.8 Type operations

IVML provides the following type-specific operations: **isTypeOf**, **isKindOf** and **typeOf**. The first two operations are similar to the related operations in OCL. The latter one returns the actual type (MetaType) of a decision variable, compound field or container element. MetaType allows equality and unequality comparisons. In addition, the containers provide the operations **typeSelect, selectByType, selectByKind** and **typeReject** which select elements from a container according to their actual type based on the **isTypeOf** operation. The **asType** operation allows re-typing and casting as long as the involved types and values are type compatible.

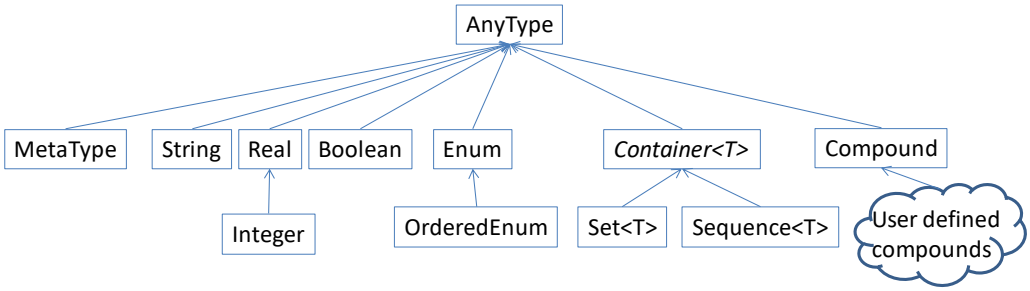

Figure 1: IVML type hierarchy

3.1.9 Side effects

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

3.1.10 Constraint variables / Named constraints

IVML allows defining constraint variables, i.e., variables of type Constraint that can hold a variable. Declaring a constraint variable looks like declaring an usual variable

```
Integer i = 10;
Constraint myConstraint = i > 10;
```

In the first line, we define the variable `i`, which is just used for illustrating a constraint. The second line defines a variable of constraint type (one can also understand this as a "named constraint"), which holds the constraint given by the default value expression, i.e., `i > 10`. IVML just treads the value of a constraint variable like an usual constraint, which may, as any other variable be changed (in contrast to usual constraints). More precisely, when evaluating the constraints of a scope, the constraints in the constraint variables of the scope are evaluated as if they are defined as usual constraints.

Using constraint variables instead of simple named constraints allows changing, freezing and even disabling certain constraints at evaluation time, in particular in importing projects. Disabling the constraint in the example above looks like

```
myConstraint = true;
```

or even more radically (setting the variable to undefined)

```
myConstraint = null;
```

As for any other variable, overriding the value of a constraint variable is allowed only once per project scope.

Of course, constraint variables can be used within compounds or in containers, i.e., in terms of containers of constraints, or in combinations with derived types.

3.1.11 Undefined values

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

3.1.12 Blocks

In specific constraint concepts, multiple constraint expressions can be combined into a sequential block, which can be used instead of an expression in specific constraint constructs. A block can be seen as syntactic sugar for a conjunctive expression over multiple assignments, but with flexibility about the result value. A block starts with a left curly bracket ('{') and ends with a right curly bracket ('}'). Constraints in a block

must end with a semicolon (';'). The type of the block and its value is determined by the last constraint in the block. We will explicitly indicate the concepts where blocks can be used.

3.1.13 If-then-else-endif Expressions

The if-then-else-endif construct supports determining a value depending on a Boolean expression, similar to distinction of cases in mathematics. Exactly one expression must be used within the `then` and `else` parts, both yielding the same type. The `else` part is not optional. Akin to implies, the condition of an if-then-else expression is not subject to value propagation. Then- or else-expression may be blocks.

```
if contents[0].type == "video"
  then contents[0].bitrate
  else contents[0].highBitrate
  endif;
```

3.1.14 Let Expressions

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

```
let Integer sumBitrate = bitrates.sum()
  in sumBitrate <= 256;
```

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

3.1.15 User-defined operations

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

```
def Integer actualBitrate(Contents c) =
   if c.type == "video"
     then c.bitrate
     else c.highBitrate
   endif;
```

The name of an operation may not conflict with keywords, types, decision variables, etc. An user-defined operation may be used similar to build-in operations. User-defined operations in imported projects are available as long as the imported project does not define interfaces. The import sequence is relevant as the first model containing the right signature will take precedence. Please note that prefix or infix use of user-defined operations is not supported.

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

User-defined operations can be overridden, i.e., operations of same name with the same number of parameters of refined parameter types and refined result type may be specified. Overridden operations are subject to dynamic dispatch during constraint evaluation, i.e., the operation with the best fitting signature to the actual parameters is selected for execution at runtime.

Typically, dynamic dispatch is rather runtime consumptive as candidate operations are also searched in imported projects and several type comparisons have to be done to find the best matching operation. As dynamic dispatch is not needed for all user-defined operations, a user-defined operation can be declared as **`static`**, which, akin to existing programming languages, leading to a static rather than a dynamic binding effectively disabling dynamic dispatch for the respective operation. The static variant of the `actualBitrate` operation shown above looks like

```
def static Integer actualBitrate(Contents c) =

   if c.type == "video"

     then c.bitrate

     else c.highBitrate

  endif;
```

User-defined operations can be annotated similar to Java, e.g.,

```
@dispatchBasis
def static Integer length(Contents c) = 1;
```

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.

3.1.16 Container operations

IVML defines many operations on the container types. These operations are specifically meant to enable a flexible and powerful way of constraining the contents of containers or projecting new containers from existing ones. Following OCL, we distinguish container operations into iterator-based (lambda-like) operations applying an expression to all elements in a container and usual container operations such as returning the size of a container. However, we support only a relevant subset of the various notations in OCL (see Section 3.6 for details). The different forms of iterator-based container operations are described in the following paragraphs. As in OCL, all container operations (and only those) can be applied using the arrow-operator '->', while usual container operations (not iterator-based container operations) akin to all other operations defined for IVML can alternatively be applied using the dot-operator '.'.

Container operations require a container 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. As a consequence, a container operation such as `closure` can be called also on a compound slot, e.g., containing a reference to another decision variable.

In the first versions of OCL, all container operations returned flattened containers, i.e. the entries of nested containers instead of the containers were taken over into the results. However, this was considered as an issue in OCL since version 2 and, in particular, does not fit to the explicit hierarchical nesting in IVML. Thus, container operations in IVML do, as OCL 2 operations, not apply flattening by default.

Sometimes an expression using operations results in a container, while we are interested only in a special subset of the container. The iterator-based container operation **`select`** specifies a subset of a `container` (here denoting an expression that either can be evaluated to a container or is turned into an implicit container):

```
container->select(boolean-expression)

container->select(t|boolean-expression-with-t)

container->select(ElementType t|

  boolean-expression-with-t)
```

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

Example:

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

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

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

As shown in the previous section, the select and reject operations always result in a sub-container of the original container. If we want to specify a container which is derived from some other container, but which contains different objects from the original container, we can use a **collect** operation. Note that as in OCL the **collect** operation flattens the result, while the **collectNested** operation keeps the nesting. The **collect** and the **collectNested** operation use the same syntax as select and reject and can be written as one of:

```
container->collect(expression)

container->collect(t|expression-with-t)

container->collect(ElementType t|expression-with-t)
```

The **sortedBy** operation can be called using the same syntax as for c**ollect**, but takes the values calculated by the expression as key to sort the values in the container the operation was called on (returning a copy of the original container).

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

```
container->forAll(boolean-expression)

container->forAll(t|boolean-expression-with-t)

container->forAll(ElementType t|

  boolean-expression-with-t)
```

Example:

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

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

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

```
container->forAll(t1, t2|

  boolean-expression-with-t1-and-t2)
```

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

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

```
container->exists(boolean-expression)

container->exists(t|boolean-expression-with-t)

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

Depending on the type of the container further related operation may be defined such as **isUnique**. Details on the operations of the individual types will be given in Sections 3.2-3.7.

If references are used, sometimes the set of all (transitively) referenced decision variables is needed, in particular in topological variability models. This set can either be calculated explicitly using recursive user-defined operations (see Section 3.1.15) or through the **closure** operation.

```
container->closure(expression)

container->closure(t|expression-with-t)

container->closure(ElementType t|expression-with-t)
```

As in OCL, the **closure** operation performs a depth first preorder traversal of the references specified by the `expression`. The operation stops visiting a potential path if the value returned by `expression` was already added to the result or if the `expression` evaluates to an empty collection. Undefined (not-configured) references will cause that the result of closure is also undefined (in contrast to `null`, which indicates that no reference is configured). Further, the operation handles single references (visit the given reference next) or collections of references (visit all given references next) returned by `expression`. As in OCL, the result of this operation is always a Set. In addition, IVML provides the **isAcyclic** operation, which can be used akin to the **closure** operation, but returns whether a cycle occurred while calculating the closure specified by the expression.

One special case of container operation is to aggregate one value over all values in a container by applying a certain expression or function. However, this comes close to the iterate operation in OCL. As we specifically target value aggregations define the

iterate (or the equivalent **apply**[9]) operation while reusing the already known syntax:

```
container->iterate(t; ResultType r = initial|
  r = expression-with-t)
container->iterate(ElementType t; ResultType r = initial|
  r = expression-with-t)
```

This operation initializes the result variable `r` with the `initial` expression and applies the `expression-with-t` to each element in the container. The result of `expression-with-t` is used to update successively the result variable. Finally, the operation returns the value of `r` after processing the last element in `container`. Please note that the result variable is always defined using a specific type which, in turn, defines the result type of the apply operation. Between the iterator `t` and the result variable, further temporary variables of different type can be defined. However, iteration happens only over the iterator (as well as following iterator variables of compatible type, which implicitly form a cross-product over the `container`).

Example 1:

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

Example 2:

```
setOf(String) orgVars;
setOf(refTo(String)) referedVars = ...;

// "Derefer" values
orgVars = referedVars->iterate(refTo(String) itr;
  setOf(String) tmpValues = {} |
tmpValues.add(refBy(itr)));
```

3.2 Foundational Types

Similar to OCL, in the IVML constraint language all operations are defined on individual IVML types and can be accessed using the "." operator, such as `set.size()`. However, this is also true for the equality, relational and mathematical operators but they are typically given in alternative infix notation, i.e. 1 + 1 instead of 1.+(1). Further, the unary negation is typically stated in prefix notation. Iterative container operations such as `forAll` are the only[10] operations in

[9] Originally, `apply` was a simplified version of `iterate`. Over the time, due to practical application, the parameters and semantics of `apply` converged against OCL `iterate`. During OCL alignment, `apply` became an alias for `iterate` and was kept for IVML compatibility, similarly `typeSelect` and `selectByType`.

[10] This is due to technical restrictions realizing IVML with Xtext.

IVML which are accessed by "->". However, IVML also defines some specific operations that are also listed with their defining type below.

Below, we denote the actual type on which an individual operation is defined as the *operand* of the operation (called *self* in OCL). The parameters of an operation are given in parenthesis. Further, similar to the declaration of decision variables in IVML, we use in this section the Type-first notation to describe the signatures of the operation. If alias operations, i.e., operations of same signature and return type but different name do exist, we list them after a slash behind the signature of the primary operation (omitting the result type of the alias operations).

Some of the types defined below are marked as internal types. These types are required to define operations and signatures, but are not intended for direct use in IVML and, thus, shall not be accessible.

3.2.1 Meta[11]

MetaType represents the actual type of an object such as a specific user-defined container.

- **Boolean == (Meta t)**
  True if the *operand* is the same as *t*.
- **Boolean <> (Meta a) / != (Meta t)**
  True if *operand* is different from *a*.
- **setOf(refTo(T)) allInstances ()**
  Returns all instances of *operand* known to the actual configuration. Let T be the actual type of *operand*, then the operation returns a set of references to T, i.e., access to the original instances rather than copies. For non-compound types, in particular basic types, the resulting set is always empty.

3.2.2 Any

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

- **Boolean == (Any a)**
  True if the *operand* is the same as *a*. This operation is interpreted as a value assertion if it is used standalone (empty implication) or on the right side of an implication. It is interpreted as an equality test if used on the left side of an implication.
- **Boolean <> (Any a) / != (Any a)**
  True if *operand* is different from *a*.
- **Boolean isDefined()**
  True if *operand* has a defined / configured value (default or explicit), false if it is not yet configured or its configuration was removed (null). Returns undefined if the *operand* was never configured.
- **Boolean forceDefined()**

[11] Internal type. used for specifying the operations only. Cannot be used within IVML.

True if *operand* has a defined / configured value (default or explicit), false if it is not yet configured or its configuration was removed (null). In contrast to `isDefined`, Returns false if the *operand* was never configured.

- **Boolean isKindOf (Meta type)**
  True if *type* is either the direct type or one of the supertypes of the actual type of the *operand*. Please note, that this operation distinguishes between reference types and underlying base types, i.e., if `v` is a variable there is a semantic difference between `v.isKindOf(MyCompound)` and `v.isKindOf(refTo(MyCompound))`.
- **Boolean isTypeOf (Meta type)**
  True if the *type* and the actual type of *operand* are the same. This operation can be seen as an alias for typeOf() == *type*. Please note, that this operation distinguishes between reference types and underlying base types, i.e., if `v` is a variable there is a semantic difference between `v.isTypeOf(MyCompound)` and `v.isTypeOf(refTo(MyCompound))`.
- **Any asType(Meta type)**
  Converts the operand to the given type. Undefined if not possible.
- **T copy(String p)**
  Creates a copy of the value of *operand*. Produces a shallow copy if *p* is empty and a deep copy, i.e., creating new variables with *p* as name prefix for the variable name in the same package in case of reference values. Undefined if *operand* is not configured.
- **String locale()**
  Returns the (global) locale via *operand* as "*language*" or "*language_country*". As in OCL, the default locale is "en_US", but may be redefined by the implementing tool, e.g., via properties.
- **String locale(String s)**
  Changes the (global) locale via *operand* and returns the actual locale. *s* shall be given as "*language*" or "*language_country*".
- **Meta typeOf ()**
  The type information of the actual type.

3.2.3 Version[11]

The version type is an internal type (actually not supported as a regular type for variables) for defining version constraints[12]. Using the type name "Version" is discouraged. Thus, the version type supports only the following operations, in particular not the type operations provided by Any:

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

[12] Actually, this is in preparation.

- **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*.

3.2.4 FreezeVariable[11]

The FreezeVariable type is an internal type just used within the but-part of a freeze block. This type supports only the following operations, in particular not the type operations defined by Any:

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

Further, a FreezeVariable provides access to all annotations defined by all elements mentioned within the freeze block. In case that different types of annotations for the same name are defined, just the first annotation definition in sequence of the elements of the freeze block is considered.

3.3 Basic Types

The following types represent basic types for configuration modelling.

3.3.1 Real

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

- **Boolean = (Real r)**
  Assigns the value *r* to the variable *operand* and returns *true*[13].
- **Boolean < (Real r)**
  True if the *operand* is less than *r*.
- **Boolean > (Real r)**
  True if the *operand* is greater than *r*.
- **Boolean <= (Real r)**
  True if the *operand* is less than or equal to *r*.
- **Boolean >= (Real r)**
  True if the *operand* is greater than or equal to *r*.
- **Real + (Real r)**
  The value of the addition of *r* and the *operand*. The operation evaluates to undefined if the evaluation is mathematically illegal.
- **Real - ()**

[13] The Boolean return type is required as stand-alone constraints must be of Boolean type. The result of an assignment operation is always `true`.

The negative value of the *operand*. The operation evaluates to undefined if the evaluation is mathematically illegal

- **Real - (Real r)**
  The value of the subtraction of *r* from the *operand*. The operation evaluates to undefined if the evaluation is mathematically illegal
- **Real * (Real r)**
  The value of the multiplication of the *operand* and *r*. The operation evaluates to undefined if the evaluation is mathematically illegal
- **Real / (Real r)**
  The value of the *operand* divided by *r*. The operation evaluates to undefined if the evaluation is mathematically illegal, in particular if *r* is equal to zero .
- **Real abs()**
  The absolute value of the *operand*.
- **Integer floor ()**
  The largest integer that is less than or equal to the *operand*[14].
- **Integer round()**
  The integer that is closest to *the operand*. When there are two such integers, the largest one[14].
- **Real min (Real r)**
  The minimum of the *operand* and *r*.
- **Real max (Real r)**
  The maximum of the *operand* and *r*.
- **String toString (Real r)**
  Returns the string representation of *operand*.

3.3.2 Integer

The standard type Integer represents the mathematical concept of integer following the Java range restrictions for integer values. Thus, Integers range from Java's Integer.MIN_VALUE (-2147483648) to Java's Integer.MAX_VALUE (2147483647). Note that as in OCL, Integer is a subclass of Real.

- **Boolean = (Integer i)**
  Assigns the value *i* to the operand and returns *true***Error! Bookmark not defined.**. A real value cannot be directly assigned to an integer, but must be converted, e.g., using the **floor** operation.
- **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*.

[14] Since IVML has no BigInteger implementation, the result will be between -2147483648 and 2147483647 (see Section 3.3.2 for more details).

- **Integer + (Integer i)**
  The value of the addition of the *operand* and *i*. The operation evaluates to undefined if the evaluation is mathematically illegal
- **Integer - ()**
  The negative value of the *operand*. The operation evaluates to undefined if the evaluation is mathematically illegal
- **Integer - (Integer i)**
  The value of the subtraction of *i* from the *operand*. The operation evaluates to undefined if the evaluation is mathematically illegal
- **Integer * (Integer i)**
  The value of the multiplication of the *operand* and *i*. The operation evaluates to undefined if the evaluation is mathematically illegal
- **Real / (Integer i)**
  The value of the *operand* divided by *i*. The operation evaluates to undefined if the evaluation is mathematically illegal, in particular if *i* is equal to zero.
- **Integer abs()**
  The absolute value of the *operand*.
- **Integer div (Integer i)**
  The number of times that *i* fits completely within the *operand*.
- **Integer max (Integer i)**
  The maximum of the *operand* and *i*.
- **Integer min (Integer i)**
  The minimum of the *operand* and *i*.
- **Integer mod (Integer i)**
  The result is the *operand* modulo *i*.
- **String toString (Integer i)**
  Returns the string representation of *operand*.

3.3.3 Boolean

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

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

- **Boolean not ()**
  True if *operand* is false and vice versa.
- **String toString (Boolean b)**
  Returns the string representation of *operand*.

3.3.4 String

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

- **Boolean = (String s)**
  Assigns the value *s* to the operand and returns *true***Error! Bookmark not defined.**.
- **Boolean < (String s)**
  Returns whether *operand* is less than *s* using the current locale.
- **Boolean <= (String s)**
  Returns whether *operand* is less than or equal to *s* using the current locale.
- **Boolean > (String s)**
  Returns whether *operand* is greater than *s* using the current locale.
- **Boolean >= (String s)**
  Returns whether *operand* is greater than or equal to *s* using the current locale.
- **String + (String s) / concat (String s)**
  The concatenation of the *operand* and *s*.
- **String at(Integer i)**
  Returns the character at index *i* of operand, with *i* in (0;size()-1)
- **SequenceOf(String) characters ()**
  Returns the characters of *operand* as a sequence of strings. Returns an empty sequence of *operand* is empty.
- **Boolean equalsIgnoreCase (String s)**
  Returns whether *operand* and *s* contain the same characters regarding the current locale ignoring case differences.
- **Integer indexOf(String s)**
  Returns the 0-based index of the first occurrence of *s* in *operand*. Returns -1 if *s* does not occur in *operand*.
- **Integer size ()**
  The number of characters in the *operand*.
- **Boolean substitutes (String r, String s)**
  Replaces all occurrences of the regular expression *r* in the *operand* by *s*. Regular expressions are given in the Java regular expression notation. For example, the following operation will substitute the occurrence of `"@"` with `"{at}"` in an e-mail-address:
  ```
  mail.substitutes("@", "{at}");
  ```
- **String substring (Integer lower, Integer upper)**
  The sub-string of the *operand* starting at character number *lower*, up to and including character number *upper*. Character numbers run from 0 to *size()*.
- **Boolean matches (String r)**

Returns whether the *operand* matches the regular expression *r*. Regular expressions are given in the Java regular expression notation. For example, the following operation will check whether `mail` is a valid e-mail-address:

```
mail.matches("[\w]*@[\w]*.[\w]*");
```

- **Boolean toBoolean ()**
  Converts the *operand* to a Boolean value. Inspired by OCL, "true" (ignoring cases based on the current locale) leads to the value true, everything else to false.
- **Integer toInteger ()**
  Converts the *operand* to an Integer value. If the operand cannot be converted, the result of the operation is undefined yielding that the containing expression is undefined (cf. Section 3.1.11).
- **String toLowerCase ()**
  Converts the *operand* to lower case characters using the current locale.
- **Real toReal ()**
  Converts the *operand* to a Real value. If the operand cannot be converted, the result of the operation is undefined yielding that the containing expression is undefined (cf. Section 3.1.11).
- **String toString ()**
  Returns the string representation of *operand*, i.e., *s*.
- **String toUpperCase ()**
  Converts the *operand* to upper case characters using the current locale.

3.4 Enumeration Types

Enumerations allow the definition of sets of named values.

3.4.1 Enum

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

- **Boolean = (Enum e)**
  Assigns the value *e* to the *operand* and returns *true***Error! Bookmark not defined.**.
- **Integer ordinal (Enum e)**
  Returns the ordinal of the given enum value represented by the *operand*. In case of ordered enums, the value is explicitly defined in the model. In case of ordinary enums, the internally assigned ordinal is returned.

3.4.2 OrderedEnum

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

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

True if the *operand* is less than or equal to the ordinal value of the literal *l*.

- **Boolean >= (OrderedEnum l)**
  True if the *operand* is greater than or equal to the ordinal value of the literal *l*.
- **OrderedEnum min(OrderedEnum l)**
  The literal of *operand* and *l* having the minimum ordinal value (including equality).
- **OrderedEnum max(OrderedEnum l)**
  The literal of *operand* and *l* having the maximum ordinal value (including equality).

3.5 Constraint

The basic type Constraint represents a constraint variable, i.e., a variable (freezable) constraint. In addition to the operations provided by Any, the Constraint type provides the following operations:

- **Boolean = (Constraint c)**
  Assigns the constraint *c* to the operand and returns *true***Error! Bookmark not defined.**.

3.6 Container Types

This section defines the operation of the container types. The two IVML containers Set and Sequence are both subtypes of the abstract container type Container. Each container type is actually a template type with one parameter. 'T' denotes the parameter. A concrete container type is created by substituting a type for the T. So a container of integers is referred in IVML by `setOf(Integer)`. Although the keyword `containerOf` does not exist, we will use it in this section to denote types of Collection (Section **Error! Reference source not found.**).

3.6.1 Container

Container is the abstract superclass of all containers in IVML.

- **T any (Iterator | expression)**
  Returns any element in the *source* container for which *expression* evaluates to true. If there is more than one element for which *expression* is true, one of them is returned. *any* may have at most one iterator variable.
- **T avg()**
  The average of all elements in the *operand*. Elements must be of a type supporting the / operation (Integer or Real).
- **containerOf(T) collect (Iterator | expression)**
  The flattened container of elements that results from applying *expression* to every member of the *source* set. `collect` may have at most one iterator variable.
- **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. `collectNested` may have at most one iterator variable.

- **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. Undefined (not-configured) references will cause that the result of closure is also undefined (in contrast to `null`, which indicates that no reference is configured). See `isAcyclic`.
- **Integer count (T o)**
  The number of times that *o* occurs in the container *operand*.
- **Boolean excludes (T o)**
  True if *object* is not an element of *o*, false otherwise.
- **Boolean excludesAll (containerOf<T> object)**
  True if *operand* contains none of the elements from *c*, false otherwise.
- **Boolean exists (Iterators | expression)**
  Results in true if *expression* evaluates to true for at least one element in the *operand* container.
- **Boolean forAll (Iterators | expression)**
  Results in true if *expression* evaluates to true for each element in the *operand* container.
- **Boolean includes (T o)**
  True if *object* is an element of *o*, false otherwise.
- **Boolean includesAll (containerOf<T> c)**
  True if *operand* contains all elements from *c*, 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. Undefined (not-configured) references will cause that the result of closure is also undefined (in contrast to `null`, which indicates that no reference is configured). See `closure`.
- **Boolean isEmpty ()**
  Is the *operand* the empty container?
- **Boolean isDefined()**
  True if *operand* has a defined / configured value (default or explicit), false if it is not yet configured or its configuration was removed (null). Returns undefined if the *operand* was never configured.
- **Boolean forceDefined()**
  True if *operand* has a defined / configured value (default or explicit), false if it is not yet configured or its configuration was removed (null). In contrast to `isDefined`, Returns false if the *operand* was never configured.
- **Boolean isUnique (Iterator | expression)**
  Results in true if *expression* evaluates to a different value for each element in the *operand* container; otherwise, result is false. *isUnique* may have at most one iterator variable.

- **R iterate (Iterator; R result | result = expression) / apply (Iterator; R result | result = expression)**
  Iterates the given expression on the operand container using the specified iterator variable and stores the result in the last variable (used here as a local variable declaration) which is returned as the result of this operation. Expression shall use the result variable for aggregating values. Apply may have at most one iterator variable and needs to specify the result variable. Examples are given on page 49.
- **T min()**
  The minimum of all elements in the *operand*. Elements must be of a type supporting the < operation (Integer or Real).
- **T min(Iterator | expression)**
  The minimum of all elements from *operand* according to expression (must be of type Real or Integer) – the generic container operation variant of min(). The expression is typically a slot projection over T or a calculation of a slot projection. The result is the (first found) element with the minimum value returned by expression. The operation is undefined if *operand* does not contain elements.
- **T max()**
  The maximum of all elements in the *operand*. Elements must be of a type supporting the > operation (Integer or Real).
- **T max(Iterator | expression)**
  The maximum of all elements from *operand* according to expression (must be of type Real or Integer) – the generic container operation variant of max(). The expression is typically a slot projection over T or a calculation of a slot projection. The result is the (first found) element with the maximum value returned by expression. The operation is undefined if *operand* does not contain elements.
- **Boolean notEmpty ()**
  Is the *operand* not the empty container?
- **Boolean one (Iterator | expression)**
  Results in true if there is exactly one element in the *operand* container for which *expression* is true. *one* may have at most one iterator variable.
- **T product()**
  The multiplication of all elements in the *operand*. Elements must be of a type supporting the * operation (Integer or Real). As usual in mathematics, the result for an empty collection is 1.
- **containerOf(T) reject (Iterator | expression)**
  The sub-container for which expression is false. *reject* may have at most one iterator variable.
- **containerOf(T) select (Iterator | expression)**
  The sub-container for which expression is true. *select* may have at most one iterator variable.
- **Integer size ()**
  The number of elements in the container *operand.*
- **containerOf(T) sortedBy (Iterator | expression)**

Returns the flattened container of elements sorted by the keys calculated by *expression* for each individual value. Comparison shall be done based on the comparison operations of the individual types or, if not defined, based on the IVML string representation applying actual locale.

- **T sum()**
  The addition of all elements in the *operand*. Elements must be of a type supporting the + operation (Integer or Real).

3.6.2 Set

The Set is the mathematical set. It contains elements without duplicates. Set inherits the operations from Container. As in OCL, Sequence supports the `sortedBy` operation introduced by Container returning a Set although a Set does not imply any sorting. For this reason, we mention the operation here but do not list its signature explicitly below.

- **Boolean = (setOf(T) s)**
  Assigns the value *s* to the operand and returns *true***Error! Bookmark not defined.**.
- **Boolean == (setOf(T) s)**
  Evaluates to true if *operand* and *s* contain the same elements.
- **Boolean <> (Any a) / != (Any a)**
  True if the *operand* is different from *a*.
- **setOf(T) - (setOf(T) s)**
  Returns the elements of *operand* that are not in *s*.
- **T add(T e)**
  Adds an element *e* to the set denoted by *operand*. If *e* was already in *operand*, nothing happens. Returns *e*.
- **setOf(T) collect (Iterator | expression)**
  The flattened container of elements that results from applying *expression* to every member of the *source* set. `collect` may have at most one iterator variable.
- **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. `collectNested` may have at most one iterator variable.
- **setOf(T) excluding (T object)**
  Returns a set containing all elements of *operand* without *object*.
- **setOf(T) flatten (setOf(containerOf(T))**
  Returns the (deep) flatten set of *operand*, i.e., all (recursively) contained nested containers are turned into individual elements and the unified set (without duplicates) is returned.
- **setOf(T) including (T object)**
  Returns a set containing all elements of *operand* plus *object*.
- **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*).
- **setOf(T) symmetricDifference (setOf(T) s)**

Returns the elements that are either in *operand* or in *s*, but not in both.

- **setOf(T) selectByKind (Meta T) / typeSelect(Meta T)**
  Results the subset of elements from *operand* which are of type *T* (including sub-types).
- **setOf(T) selectByType (Meta T)**
  Results the subset of elements from *operand* which are of type *T*.
- **sequenceOf(T) toSequence() / asSequence ()**
  A Sequence that contains all the elements from *operand,* in undefined order.
- **setOf(T) toSet() / asSet ()**
  A Set identical to *operand*. This operation exists for convenience reasons.
- **setOf(T) typeReject (Meta T)**
  Results the subset of elements from *operand* which are not of type *T*.
- **setOf(T) union (setOf(T) s)**
  The union of *operand* and *s*.

3.6.3 Sequence

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

- **Boolean = (sequenceOf(T) s)**
  Assigns the value *s* to the operand and returns *true***Error! Bookmark not defined.**.
- **Boolean == (sequenceOf(T) s)**
  Evaluates to true if *operand* and *s* contain the same elements.
- **T [Integer i] / T at (Integer i)**
  The *i-th* element of the sequence *operand*. Valid indices run from 0 to *size()-1*.
- **T add(T e)**
  Adds an element *e* to the end of the sequence denoted by *operand*. If *e* was already in *operand*, nothing happens. Returns *e*.
- **sequenceOf(T) append (T object)**
  The sequence of elements, consisting of all elements of *operand*, followed by *object*.
- **sequenceOf(T) collect (Iterator | expression)**
  The flattened container of elements that results from applying *expression* to every member of the *source* set. `collect` may have at most one iterator variable.
- **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. `collectNested` may have at most one iterator variable.
- **sequenceOf(T) excluding (T object)**
  Returns a sequence containing all elements of *operand* without *object*.
- **T first ()**
  The first element in *operand*.
- **Boolean hasDuplicates()**
  Returns whether at least one of the elements in *operand* has a duplicate.

- **sequenceOf(T) including (T object)**
  Returns a sequence containing all elements of *operand* plus *object*.
- **Integer indexOf(T object)**
  The index of object *object* in the sequence *operand*. -1 if *object* is not a member of *operand*.
- **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*.
- **Boolean isSubsequenceOf(sequenceOf(U) o)**
  Returns whether *operand* is a subsequence (considering the sequence and including equality) of *o*.
- **T last()**
  The last element in *operand*.
- **sequenceOf(T) reverse()**
  Returns a sequence containing the reversed sequence of *operand*.
- **sequenceOf(T) prepend(T object)**
  The sequence consisting of *object*, followed by all elements in *operand*.
- **sequenceOf(T) selectByKind (Meta t) / typeSelect (Meta t)**
  Results the subset of elements from *operand* which are of type *t* (including sub-types)
- **sequenceOf(T) selectByType (Meta t)**
  Results the subset of elements from *operand* which are of type *t*.
- **sequenceOf(T) sortedBy (Iterator | expression)**
  Returns the flattened container of elements sorted by the keys calculated by *expression* for each individual value. Comparison shall be done based on the comparison operations of the individual types or, if not defined, based on the IVML string representation applying actual locale.
- **sequenceOf(T) flatten (sequenceOf(containerOf(T))**
  Returns the (deep) flatten sequence of *operand*, i.e., all (recursively) contained nested containers are turned into individual elements and the unified sequence (including duplicates) is returned.
- **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 ()**
  The Set containing all the elements from *operand*, with duplicates removed.
- **sequenceOf(T) toSequence() / asSequence ()**
  The Sequence identical to the *operand* itself. This operation exists for convenience reasons.
- **sequenceOf(T) typeReject (Meta t)**
  Results the subset of elements from *operand* which are not of type *t*.
- **Boolean overlaps(sequenceOf(U) o)**
  Returns the sequence that *operand* and *o* have elements in common.
- **Sequence(T) union (sequenceOf(T) s)**

The union of *operand* and *s*.

3.7 Compound Types

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

- **Boolean = (Compund c)**
  Assigns the value *c* to the operand and returns *true***Error! Bookmark not defined.**.
- **Boolean isDefined()**
  True if *operand* has a defined / configured value (default or explicit), false if it is not yet configured or its configuration was removed (null). Returns undefined if the *operand* was never configured.
- **Boolean forceDefined()**
  True if *operand* has a defined / configured value (default or explicit), false if it is not yet configured or its configuration was removed (null). In contrast to `isDefined`, Returns false if the *operand* was never configured.

4

4 Implementation Status

The realization of IVML and IVML-related tools is still in progress. In this section, we summarize the status as it is implemented by the recent version of EASy-Producer. This encompasses the IVML model/parser, the discontinued Drools reasoning support, the actual Configuration (initialization) and the EASy reasoner. We will first indicate the support for core IVML concepts in Table 1, then for advanced concepts in Table 2.

IVML concept	IVML model, parser, semantic analyzer	EASy reasoning support (Drools)	EASy Configuration support	EASy IVML reasoning support
project	x	(x)	x	x
Boolean	x	x	x	x
integer	x	narrowing interval values	x	x[15]
real	x	x	x	x
string	x	?	x	x
enumerations	x	x	x	x
container	x	?	x	x[16]
type derivation and restriction	x	x	x	x
compounds	x	x	x	x
null values	x	-	x	x
decision variables	x	x	x	x
constraints	x	normal, custom type constraints, compound constraints	x (not in UI)	x
constraints as variables	x	-	x (not in UI)	x
configurations	x	initial values	x	x

Table 1: Implemented IVML core concepts (x=full support, -=no support as not responsible, partial support indicated by text)

[15] Currently, no narrowing of variables sufficiently limited by constraints. Future reasoning chain may target this.

[16] Future reasoning chain may create instances as part of propagation.

IVML concept	IVML model, parser, semantic analyzer	EASy reasoning support (Drools)	EASy Configuration support	EASy IVML reasoning support
configurations	x	initial values	x	x
annotations	x	-	x	x
extended compounds	x	?	x	x
referenced elements	x	-	x	x
project versioning	x	-	x	x
project composition	x	correct recursive processing missing	x (predecessor locations not passed to IVML)	x
project interfaces	x	-	x	(x)[17]
partial configuration	x	x	x	x
freezing configurations	x	x	x	x
partial evaluation	x	correct recursive processing missing	-	x

Table 2: Implemented IVML advanced concepts (x=full support, -=no support as not responsible, partial support indicated by text)

The implementation of the IVML language utilizes Eclipse xText for generating parsers and editor support. Using IVML 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 IVML 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.

[17] In development, may need partial freezing.

5 IVML Grammar

In this section we depict the actual grammar for IVML. The grammar is given in six sections (basic modeling concepts, basic types and values, advanced modeling concepts, basic constraints, advanced constraints and terminals) in terms of a simplified xText[18] grammar (close to ANTLR[19] or EBNF). Simplified means, that we omitted technical details used in xText to properly generate the underlying EMF model as well as trailing ";" of the ANTLR grammar specification language (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. Further, some concepts imply alternative styles, e.g., qualified names may include '.' or '::' for qualification and non-iterating container operations may be called by '.' or '->'. An implementation may provide a compliance mode to warn about OCL compliance if '.' is used.

5.1 Basic modeling concepts

```
VariabilityUnit:
      Project*

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

ProjectContents:
      (Typedef
            | VariableDeclaration
            | Freeze
            | Eval
            | ExpressionStatement
            | AnnotateTo
            | OpDefStatement
            | AttrAssignment
```

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

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

```
      )*


ExpressionBlock:
      '{'
            ExpressionStatement+
      '}' ';'?

Typedef:
      TypedefEnum
      | TypedefCompound
      | TypedefMapping

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

TypedefEnumLiteral:
      Identifier ('=' NumValue)?

TypedefCompound:
      'abstract'? 'compound' Identifier ('refines' Identifier)?
      '{'
            (VariableDeclaration
                  | ExpressionStatement
                  | AttrAssignment
                  | Eval)*
      '}' ';'?

TypedefMapping:
      'typedef' Identifier Type TypedefConstraint? ';'

TypedefConstraint:
      'with' '(' Expression ')'

VariableDeclaration:
```

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

VariableDeclarationPart:
      Identifier ('=' Expression)?

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

5.2 Basic types and values

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

Type:
      BasicType
      | QualifiedName
      | DerivedType

NumValue:
      NUMBER

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

AccessName:
      ('.' Identifier)+

Value:
      NumValue
      | STRING
```

```
      | QualifiedName
      | ('true' | 'false')
      | Type
      | VERSION
```

5.3 Advanced modeling concepts

```
AnnotateTo :
      ('annotate' | 'attribute') Type VariableDeclarationPart
      'to' ('.' | (Identifier (',' Identifier)*)) ';'

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

AttrAssignmentPart:
    Identifier '=' LogicalExpression

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

FreezeStatement:
      '.' | (QualifiedName AccessName?)

FreezeButList:
      '(' FreezeButExpression (',' FreezeButExpression)* ')'

FreezeButExpression:
      QualifiedName AccessName? '*'?

Eval:
      'eval' '{'
          Eval*
          ExpressionStatement*
      '}'
```

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

Export:
      'export' QualifiedName (',' QualifiedName)* ';'
ImportStmt:
      'import' Identifier ('::' Identifier)? ('with' Expression)? ';'

ConflictStmt:
      'conflicts' Identifier ('with' Expression)? ';'

VersionStmt:
      'version' VERSION ';'
```

5.4 Basic constraints

```
ExpressionStatement:
      Expression ';'

Expression:
      LetExpression
      | ImplicationExpression
      | ContainerInitializer

ImplicationExpression:
      AssignmentExpression ImplicationExpressionPart*

ImplicationExpressionPart:
      ImplicationOperator AssignmentExpression

ImplicationOperator:
      'implies' | 'iff'

AssignmentExpression:
    LogicalExpression AssignmentExpressionPart?

AssignmentExpressionPart:
```

```
    '=' (LogicalExpression | ContainerInitializer)

LogicalExpression:
      EqualityExpression LogicalExpressionPart*

LogicalExpressionPart:
      LogicalOperator EqualityExpression

LogicalOperator:
      'and' | 'or' | 'xor'

EqualityExpression:
    RelationalExpression EqualityExpressionPart?

EqualityExpressionPart:
    EqualityOperator (RelationalExpression | ContainerInitializer)

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:
      (FeatureCall Call* ExpressionAccess?)
      | PrimaryExpression

Call:
      '.' FeatureCall
      | '->' ContainerOp
      | '[' Expression ']'

FeatureCall:
      (Identifier|RelationalOperator|AdditiveOperator|
       MultiplicativeOperator| EqualityOperator|ImplicationOperator|
       LogicalOperator| "not")
      '(' ActualArgumentList? ')'

ContainerOp:
      Identifier
      '(' Declarator? ActualArgumentList? ')'

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

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

ActualArgumentList:
```

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

ExpressionAccess:
      '.' Identifier Call* ExpressionAccess?

PrimaryExpression:
      (
            Literal
            | '(' Expression ')'
            | IfExpression
            | 'refBy' '(' Expression ')'
      )
      Call*
      ExpressionAccess?

ContainerInitializer:
      QualifiedName?
      '{'
            ExpressionList?
      '}'

ExpressionList:
      ExpressionListEntry (',' ExpressionListEntry)* ','?

ExpressionListEntry:
      (Identifier ('.' Identifier)? '=')?
      (ImplicationExpression | ContainerInitializer)

Literal:
      Value
```

5.5 Advanced constraints

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

IfExpression:
```

```
      'if' Expression 'then' OptBlocExpression
      'else' OptBlockExpression 'endif'

OpDefStatement:
      AnnotationDeclarations?
      'def' 'static'? Type Identifier '(' OpDefParameterList ')'
      '=' (Expression ';' | BlockExpression)

AnnotationDeclarations:
    {AnnotationDeclarations}
    ('@' Identifier)*

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

OpDefParameter:
      Type Identifier ('=' Expression)?

OptBlockExpression:
      Expression | BlockExpression

BlockExpression:
      '{' ExpressionStatement+ '}'
```

5.6 Terminals

```
Identifier:
    ID | VERSION | EXPONENT | 'version'

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] K. Bak, K. Czarnecki, and A. Wasowski. Feature and Meta-models in Clafer: Mixed, Specialized, and Coupled. In B. Malloy, S. Staab, and M. van den Brand, editors, *Proceedings of the 3rd International Conference on Software Language Engineering (SLE '10)*, volume 6563 of *Lecture Notes in Computer Science*, pages 102–122. Springer, 2010.

[2] Q. Boucher, A. Classen, P. Faber, and P. Heymans. Introducing TVL, a Text-Based Feature Modelling Language. In *Proceedings of the 4th International Workshop on Variabilits Modelling of Software-Intensive Systems (VaMoS '10)*, pages 159–162, 2010.

[3] E. S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, editors. *Software Product Lines - 15th International Conference, SPLC 2011, Munich, Germany, August 22-26, 2011*. IEEE, 2011.

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

[5] Object Management Group, Inc. (OMG). Unified Modeling Language: Superstructure version 2.1.2. Specification v2.11 2007-11-02, Object Management Group, November 2007. Available online at: http://www.omg.org/docs/formal/2007-11-02.pdf.

[6] Object Management Group, Inc. (OMG). Common Variability Language (CVL), 2010. OMG initial submission. Available on request.

[7] Mark-Oliver Reiser. Core Concepts of the Compositional Variability Management Framework (CVM). Technical Report 2009/16, Technische Universität Berlin, 2009. Available online at http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/.

[8] M. Rosenmüller and N. Siegmund. Automating the Configuration of Multi Software Product Lines. In *Proceedings of the 4th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS '10)*, pages 123–130, 2010.

[9] Rob van Ommering. *Building Product Populations with Software Components*. PhD thesis, University of Groningen, 2004.