**Date:** August 13, 2012

# Common Variability Language (CVL)

*OMG Revised Submission*

OMG document: ad/2012-08-05


**Submitters**

IBM

Fraunhofer FOKUS

Thales

Tata Consultancy Services

**Supporters**

SINTEF
University of Oslo
Tecnalia Research & Innovation
University of Waterloo
IT University of Copenhagen
INRIA
CEA
Atego
Pure-systems

**Primary Contact:**
Øystein Haugen, SINTEF
oystein.haugen@sintef.no

## USE OF SPECIFICATION - TERMS, CONDITIONS & NOTICES

The material in this document details an Object Management Community specification in accordance with the terms, conditions and notices set forth below. This document does not represent a commitment to implement any portion of this specification in any company's products. The information contained in this document is subject to change without notice.

## LICENSES

The companies listed above have granted to the Object Management Community, Inc. (OMG) a nonexclusive, royalty-free, paid up, worldwide license to copy and distribute this document and to modify this document and distribute copies of the modified version. Each of the copyright holders listed above has agreed that no person shall be deemed to have infringed the copyright in the included material of any such copyright holder by reason of having used the specification set forth herein or having conformed any computer software to the specification.

Subject to all of the terms and conditions below, the owners of the copyright in this specification hereby grant you a fully-paid up, non-exclusive, nontransferable, perpetual, worldwide license (without the right to sublicense), to use this specification to create and distribute software and special purpose specifications that are based upon this specification, and to use, copy, and distribute this specification as provided under the Copyright Act; provided that: (1) both the copyright notice identified above and this permission notice appear on any copies of this specification; (2) the use of the specifications is for informational purposes and will not be copied or posted on any network computer or broadcast in any media and will not be otherwise resold or transferred for commercial purposes; and (3) no modifications are made to this specification. This limited permission automatically terminates without notice if you breach any of these terms or conditions. Upon termination, you will destroy immediately any copies of the specifications in your possession or control.

## PATENTS

The attention of adopters is directed to the possibility that compliance with or adoption of OMG specifications may require use of an invention covered by patent rights. OMG shall not be responsible for identifying patents for which a license may be required by any OMG specification, or for conducting legal inquiries into the legal validity or scope of those patents that are brought to its attention. OMG specifications are prospective and advisory only. Prospective users are responsible for protecting themselves against liability for infringement of patents.

## GENERAL USE RESTRICTIONS

Any unauthorized use of this specification may violate copyright laws, trademark laws, and communications regulations and statutes. This document contains information which is protected by copyright. All Rights Reserved. No part of this work covered by copyright herein may be reproduced or

used in any form or by any means–graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems–without permission of the copyright owner.

## DISCLAIMER OF WARRANTY

WHILE THIS PUBLICATION IS BELIEVED TO BE ACCURATE, IT IS PROVIDED "AS IS" AND MAY CONTAIN ERRORS OR MISPRINTS. THE OBJECT MANAGEMENT GROUP AND THE COMPANIES LISTED ABOVE MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS PUBLICATION, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF TITLE OR OWNERSHIP, IMPLIED WARRANTY OF MERCHANTABILITY OR WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OR USE. IN NO EVENT SHALL THE OBJECT MANAGEMENT GROUP OR ANY OF THE COMPANIES LISTED ABOVE BE LIABLE FOR ERRORS CONTAINED HEREIN OR FOR DIRECT, INDIRECT, INCIDENTAL, SPECIAL, CONSEQUENTIAL, RELIANCE OR COVER DAMAGES, INCLUDING LOSS OF PROFITS, REVENUE, DATA OR USE, INCURRED BY ANY USER OR ANY THIRD PARTY IN CONNECTION WITH THE FURNISHING, PERFORMANCE, OR USE OF THIS MATERIAL, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

The entire risk as to the quality and performance of software developed using this specification is borne by you. This disclaimer of warranty constitutes an essential part of the license granted to you to use this specification.

## RESTRICTED RIGHTS LEGEND

Use, duplication or disclosure by the U.S. Government is subject to the restrictions set forth in subparagraph © (1) (ii) of The Rights in Technical Data and Computer Software Clause at DFARS 252.227-7013 or in subparagraph ©(1) and (2) of the Commercial Computer Software – Restricted Rights clauses at 48 C.F.R. 52.227-19 or as specified in 48 C.F.R. 227-7202-2 of the DoD F.A.R. Supplement and its successors, or as specified in 48 C.F.R. 12.212 of the Federal Acquisition Regulations and its successors, as applicable. The specification copyright owners are as indicated above and may be contacted through the Object Management Community, 250 First Avenue, Needham, MA 02494, U.S.A.

## TRADEMARKS

The OMG Object Management Community Logo®, CORBA®, CORBA Academy®, The Information Brokerage®, XMI® and IIOP® are registered trademarks of the Object Management Community. OMG™, Object Management Community™, CORBA logos™, OMG Interface Definition Language (IDL)™, The Architecture of Choice for a Changing World™, CORBAservices™, CORBAfacilities™, CORBAmed™, CORBAnet™, Integrate 2002™, Middleware That's Everywhere™, UML™, Unified Modeling Language™, The UML Cube logo™, MOF™, CWM™, The CWM Logo™, Model Driven Architecture™, Model Driven Architecture Logos™, MDA™, OMG Model Driven Architecture™, OMG MDA™ and the XMI Logo™ are trademarks of the Object Management Community. All other products or company names mentioned are used for identification purposes only, and may be trademarks of their respective owners.

## COMPLIANCE

The copyright holders listed above acknowledge that the Object Management Community (acting itself or through its designees) is and shall at all times be the sole entity that may authorize developers, suppliers and sellers of computer software to use certification marks, trademarks or other special designations to indicate compliance with these materials.

Software developed under the terms of this license may claim compliance or conformance with this specification if and only if the software compliance is of a nature fully matching the applicable compliance points as stated in the specification. Software developed only partially matching the applicable compliance

points may claim only that the software was based on this specification, but may not claim compliance or conformance with this specification. In the event that testing suites are implemented or approved by Object Management Community, Inc., software developed using this specification may claim compliance or conformance with the specification only if the software satisfactorily completes the testing suites

OMG's Issue Reporting Procedure

All OMG specifications are subject to continuous review and improvement. As part of this process we encourage readers to report any ambiguities, inconsistencies, or inaccuracies they may find by completing the Issue Reporting Form listed on the main web page *http://www.omg.org* , under Documents, Report a Bug/Issue (http://www.omg.org/technology/agreement.htm).

# Table of Contents

# PREFACE

This specification describes the Common Variability Language.

The CVL Submission Team has consisted of participants with different perspectives. We have had industry involvement both on the vendors (IBM, Atego and pure-systems) and users (THALES, CEA and TCS) sides. Furthermore we have had strong support from academia (Universities of Waterloo and Oslo, IT University of Copenhagen) and major European research companies (SINTEF, Fraunhofer, INRIA and Tecnalia Research & Innovation). Together we cover most facets of creating this language.

Even though we have put a considerable amount of effort into this document, we are still in progress.

## Intended Audience

This document is intended for architects, analysts and designers who will be specifying product lines. The document is also intended for vendors who will implement tools for CVL. It assumes some knowledge of MOF and eCore.

## About the Object Management Community

Founded in 1989, the Object Management Community, Inc. (OMG) is an open membership, not-for-profit computer industry standards consortium that produces and maintains computer industry specifications for interoperable, portable and reusable enterprise applications in distributed, heterogeneous environments. Membership includes Information Technology vendors, end users, government agencies, and academia.

OMG member companies write, adopt, and maintain its specifications following a mature, open process. OMG's specifications implement the Model Driven Architecture® (MDA®), maximizing ROI through a full-lifecycle approach to enterprise integration that covers multiple operating systems, programming languages, middleware and networking infrastructures, and software development environments. OMG's specifications include: UML® (Unified Modeling Language™); CORBA® (Common Object Request Broker Architecture); CWM™ (Common Warehouse Metamodel); and industry-specific standards for dozens of vertical markets.

More information on the OMG is available at *http://www.omg.org/*.

## OMG Specifications

As noted, OMG specifications address middleware, modeling, and vertical domain frameworks. A catalog of all OMG Specifications Catalog is available from the OMG website at:

http://www.omg.org/technology/documents/spec_catalog.htm

Specifications within the Catalog are organized by the following categories:

### OMG Modeling Specifications

- UML
- MOF
- XMI
- CWM
- Profile specifications.

### OMG Middleware Specifications

- CORBA/IIOP
- IDL/Language Mappings
- Specialized CORBA specifications
- CORBA Component Model (CCM).

### Platform Specific Model and Interface Specifications

- CORBAservices
- CORBAfacilities
- OMG Domain specifications
- OMG Embedded Intelligence specifications
- OMG Security specifications.

### Modernization Specifications •

- KDM

### Platform Independent Model (PIM), Platform Specific Model (PSM) and Interface Specifications

- CORBAservices Specifications
- CORBAfacilities Specifications
- OMG Domain Specifications
- OMG Embedded Intelligence Specifications
- OMG Security Specifications

All of OMG's formal specifications may be downloaded without charge from our website. (Products implementing OMG specifications are available from individual suppliers.) Copies of specifications, available in PostScript and PDF format, may be obtained from the Specifications Catalog cited above or by contacting the Object Management Community, Inc. at:

OMG Headquarters
250 First Avenue
Suite 100
Needham, MA 02494
USA
Tel: +1-781-444-0404
Fax: +1-781-444-0320
Email: *pubs@omg.org*

Certain OMG specifications are also available as ISO standards. Please consult *http://www.iso.org*

## Typographical Conventions

The type styles shown below are used in this document to distinguish programming statements from ordinary English. However, these conventions are not used in tables or section headings where no distinction is necessary.

Times/Times New Roman - 10 pt.: Standard body text

**Helvetica/Arial - 10 pt. Bold:** CVL metaclasses, stereotypes and other syntax elements.

**`Courier - 10 pt. Bold:`** Programming language elements.

Helvetica/Arial - 10 pt: Exceptions

**Note** – Terms that appear in *italics* are defined in the glossary. Italic text also represents the name of a document, specification, or other publication.

## Issues

The reader is encouraged to report any technical or editing issues/problems with this specification to *http://www.omg.org/ technology/agreement.htm*.

# GENERAL INFORMATION

# 1 Scope

The Common Variability Language (CVL) defined in this document is a domain-independent language for specifying and resolving variability. It facilitates the specification and resolution of variability over any instance of any language defined using a MOF-based meta-model.

# 2 Compliance Levels for CVL

CVL defines two compliance levels of conformance to the standard: a *basic compliance level* and a *high compliance level*.

The basic compliance level includes the constructs defined in this document except: variability classifiers (VClassifiers), composite variability specifications (CVSpecs), variability interfaces (VInterfaces), constraints involving classifiers and/or quantification, variability instances (VInstances), variability configurations (VConfigurations), fragment substitution variation point, opaque variation point, configurable unit variation point, and configurable unit usage variation point.

The high compliance level includes all constructs defined in this document.

# 3 Normative References

The following normative documents contain provisions which, through reference in this text, constitute provisions of this specification. Refer to the OMG site for subsequent amendments to, or revisions of, any of these publications.

- MOF v2.4.1 Specification OMG document number formal/2011-08-07
- MOF 2.0/XMI Mapping v2.1.1 Specification OMG document number ptc/2011-09-13

# 4 Definition of CVL Concepts

This section provides definitions for the central concepts of CVL.

### Variation point

Variation points are specifications of concrete variability in the base model and are part of variability models. They define specific modifications to be applied to the base model during materialization. Variation points refer to base model elements via base model handles and are bound to VSpecs. Binding a variation point to a VSpec means that the application of the variation point to the base model during materialization depends on the resolution for the VSpec. The nature of the dependency is specific to the kind of variation point.

## Substitution (variation point)

Substitution is an umbrella term for certain kinds of variation points whose application involve substituting an element for another in the base model. Substitution variation points include object substitution, link-end substitution, their parametric versions, and fragment substitution.

## Existence (variation point)

Existence is an umbrella term for certain kinds of variation points whose (negative) application involves deleting elements from the base model. They are used to express optional elements in the base model. Existence variation points include object existence, link existence, and value existence.

## Value assignment (variation point)

Value assignment is an umbrella term for certain kinds of variation points whose application involves inserting a value into a slot of the base model. Value assignment variation points include slot value assignment and parametric slot value assignment.

## Opaque variation point

An opaque variation point is a variation point whose impact on the base model is user-defined and not pre-defined in CVL. The impact is specified using a model transformation language such as QVT.

## Choice variation point

A choice variation point is a variation point which may be bound to a choice. During materialization the decision resolving the choice determines whether or not the variation point will be applied.

## Parametric variation point

A parametric variation point is a variation point that depends on a parameter and must be bound to a variable. During materialization the value supplied as the resolution for the variable is used for the parameter.

## Repeatable variation point

A repeatable variation point is a variation point that may be applied several times during materialization. It may be bound to a VClassifier and is applied once for each instance of it. Fragment substitution is a repeatable variation point.

## Variability specification (Vspec)

VSpecs are specifications of abstract variability and are part of variability models. They can be organized in trees structures representing logical constraints on their resolutions. VSpecs can have variation points bound to them. To materialize a base model with a variability model over it, resolutions for the VSpecs must be provided.

## Choice

A choice is VSpec whose resolution requires a yes/no decision (True/False). When a variation point is bound to a choice, the decision resolving that choice determines whether or not the variation point will be applied during materialization.

## Variable

A variable is a VSpec whose resolution requires providing a value of its specified type. When a parametric variation point is bound to a variable, the value provided for the variable as resolution will be used as the actual parameter when applying the variation point during materialization.

## Variability classifier (VClassifier)

A VClassifier is a VSpec whose resolution requires instantiating it zero or more times and then resolving its sub-tree for each instance. When a repeatable variation point is bound to a VClassifier it will be applied once for each instance of the VClassifier during materialization.

## Composite VSpec (CVSpec)

A CVSpec is VSpec whose resolution requires resolving the VSpecs in its type, which is a VInterface. When a configurable unit is bound to a CVSpec, its resolution determines the transformations to be applied to the internals of the unit during materialization.

## Constraint

A constraint is a logical formula or expression over VSpecs used to restrict the allowed resolutions. Constraints may be defined globally or in the context of a VSpec.

## Variability specification derivation (Vspec derivation)

A Vspec derivation is a specification how to derive a resolution for a given Vspec out of resolutions for other VSpecs. When a Vspec derivation is specified for a VSpec the resolution model need not specify a resolution for it as it is calculated according to the VSpec derivation.

## Variability specification resolution (Vspec resolution)

A VSpec resolution resolves a VSpec as part of a resolution model. Each kind of VSpec has its own kind of resolution. Choices are resolved by deciding them negatively or positively. Variables are resolved by providing a value. Classifiers are resolved by instantiating them. CVSpecs are resolved by providing a VConfiguration which contains resolutions for the VSpecs in the VInterface which is the type of the CVSpec. When the VSpecs of a variability model are organized in tree structures the VSpec resolutions for them are also organized in tree structures mirroring that of the VSpecs.

## Variability interface (Vinterface)

A VInterface is a collection of VSpecs, possibly organized in tree structures, which serves to specify what it takes to materialize a configurable unit. Each Configurable unit must be bound to a CVSpec typed by a VInterface. The VInterface may be thought of as the "variability type" or "configuration type" of the configurable unit. A VInterface can be declared as the configuration type of several configurable units since several CVSpecs may be typed by the same VInterface.

## Variability configuration (VConfiguration)

A VConfiguration is a VSpec resolution for CVSpecs. It contains resolutions for the VSpecs of the VInterface typing the CVSpec and is used to materialize a configurable unit.

### Variability model

A variability model is a collection of variation points, VSpecs, and constraints used to specify variability over a base model.

### Resolution model

A resolution model is a collection of VSpec resolutions resolving the VSpecs of a variability model.

### Base model handle

A base model handle is the CVL mechanism used to refer into the base model from variation points. There are two kinds of model handles: object handles reference objects in the base model and link handles reference links.

### Materialization

Materialization is the process of transforming a base model into a product model by applying variation points. Materialization is driven by a resolution model which provides resolutions for the Vspecs defined against the base model.

### Configurable unit (CU)

A configurable unit is a grouping of variation points associated with a base model container. It facilitates the specification of reusable components and also allows CVL to re-iterate the modular structure of the base model design. A configurable unit hides its internals and exposes a variability interface through which it may be configured. Technically, a configurable unit is a variation point that may contain other variation point, i.e. it is a composite variation point.

### Base Model

A base model is a model on which variability is defined using CVL. The base model is not part of CVL and can be an instance of any metamodel defined via MOF.

# 5 Symbols

There are no symbols defined for this specification.

# 6 Additional Information

## 6.1 Changes to Other Standards

Currently, no changes to other standards are required.`

## 6.2 How to Read this Document

This document consists of several chapters. Chapter 7 gives an overview of our CVL language. Most readers will want to read that chapter first. The reader interested in details of the metamodel should dive into Chapter 11 where the metamodel diagrams form the first sections and a manual generated from the metamodel makes up latter part**.** Finally, the more formally inclined will find evidence that we have also dedicated ourselves to the semantics. The semantics of our CVL is found in Appendix Chapter 16. The main terms are described in Chapter 4.

Those interested in how we have solved the requirements of the RFP should read Chapters 12 to 14.

## 6.3 Standards Bodies Involved

There are no other standardization bodies involved in CVL so far but some initial contact has been made with ISO/IEC JTC 1/SC 7.

## 6.4 Proof of Concept

The following tools have been applied in our experimentation with CVL:

- CVL Tool by SINTEF/UiO
- ESI PLUM by ESI
- Kermeta by INRIA
- Document production tool by Fraunhofer FOKUS

## 6.5 Acknowledgements

The following companies submitted and/or supported parts of this specification:

**Submitters**
IBM
Fraunhofer FOKUS
Thales
Tecnalia Research & Innovation (ICT- European Software Institute)
Tata Consultancy Services

**Supporters**

SINTEF
University of Oslo
University of Waterloo
IT University of Copenhagen
INRIA
CEA
Atego
pure-systems

In particular, the CVL Submission Team would like to acknowledge the participation and contribution from the following individuals (in alphabetic order by first name):

Andreas Korff, Andreas Svendsen, Andrzej Wąsowski, Birger Møller-Pedersen, Carmen Alonso, Danilo Beuche, Eran Gery, Jean-Marc Jezequel, Jerome Le Noir, João Bosco Ferreira Filho, Julia Rubin, Kacper Bak, Krzysztof Czarnecki, Marie Gouyette, Michael Wagner, Olivier Barais, Ran Rinat, Souvik Barat, Suman Roychoudhury, Vinay Kulkarni, Xiaorui Zhang, Øystein Haugen.

# CVL SPECIFICATION INTRODUCTION

# 7 Common Variability Language (CVL) Overview

The Common Variability Language (CVL) defined in this document is a domain-independent language for specifying and resolving variability. It facilitates the specification and resolution of variability over any instance of any language defined using a MOF-based meta-model. The instance over which the variability is specified and resolved is referred to as the *base model*.

This Chapter provides an overview of CVL.

## 7.1 Introduction

The Request for Proposals defined how the Common Variability Language is combined with other modeling languages as shown in Figure 1.

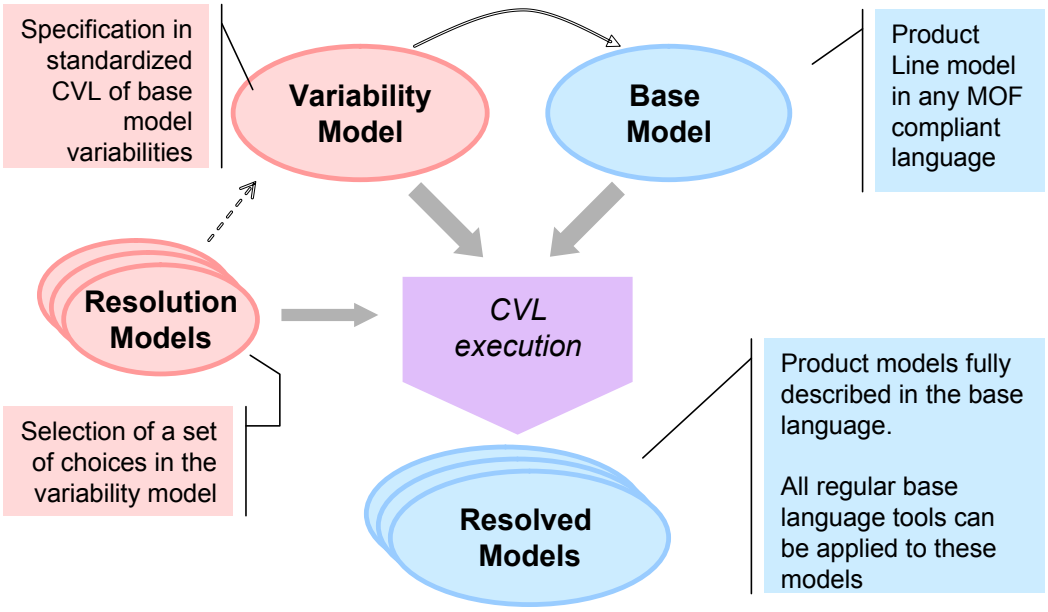



**Figure 1 Common Variability Language as specified in the Request for Proposals**

The Variability Model and the Resolution Models are defined in CVL, while the Base Model and Resolved Models can be defined in any MOF-defined language. This makes CVL a generic language, which can work in many different contexts. The RFP also requires that CVL should be executable such that the resolved models can be automatically produced from the variability model, the resolution models and the base model.

We illustrate the CVL process through a small example that uses basic constructs. Figure 2 shows a SYSML model of a printer with variability. The variability is expressed using four CVL variation points, shown at the top, referring into the base model. Three of the variation points are of kind "object existence". They indicate that model elements are optional, i.e. may or may not be present in any given product. So the part **highSpeedConnector**, the part **emgSupply** (representing emergency power supply), and the right port on the **Printer** block (used to connect **emgSupply**) are all optional. The forth variation point, of kind "slot value assignment", indicates that the value of **threshold** inside the block **EmgPower** is variable.

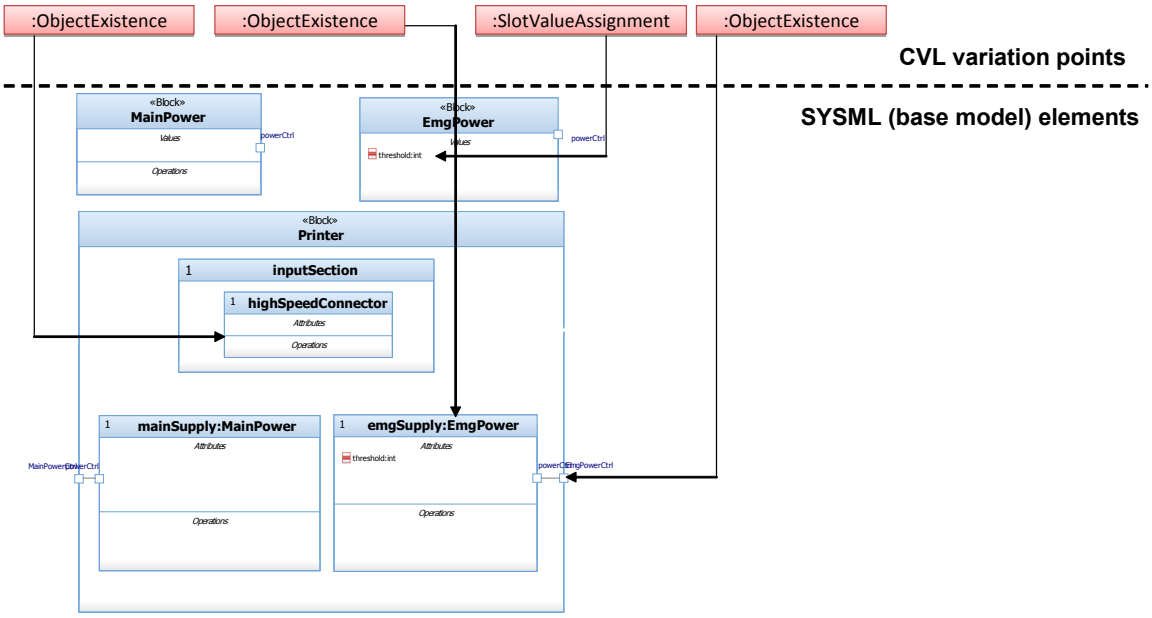



**Figure 2 - Variation points over a base model**

We designed this system knowing that in any given printer there will either be or not be an emergency power supply function, and there will either be or not be a high speed connector. We express this with a CVL *variability specification tree* (VSPec tree). This is shown in Figure 3.

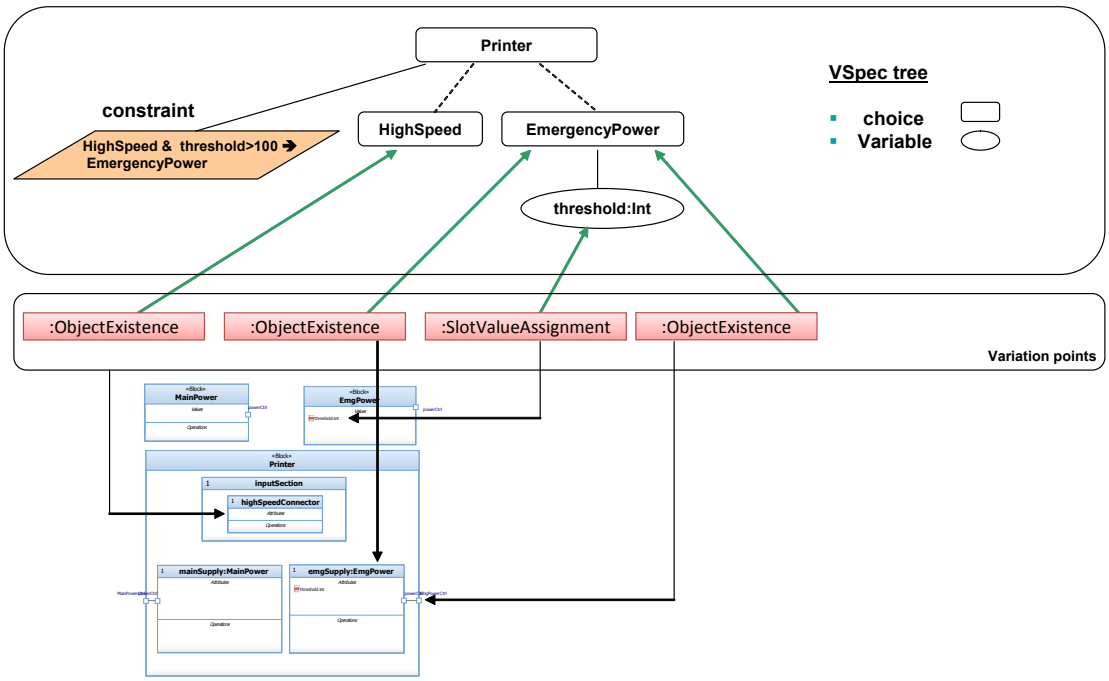



**Figure 3 - A base model with variation points bound to VSpecs**

At the top of Figure 3, the root **Printer** has two children, **HighSpeed** and **EmergencyPower,** which in CVL are called *choices*. In a resolution model a choice is *decided* positively (True) or negatively (False). On the left there is also a constraint on valid resolutions[1].

The "object existence" variation points against the SysML part **emgPower** and the right port on **Printer** both refer to the choice **EmergencyPower**, meaning both will exist only when **EmergencyPower** is decided positively. The "object existence" variation point against the SysML part **highSpeedConnector** refers to the choice **HighSpeed**. In CVL we call these mappings from a variation point to a VSpec *binding*: a variation point is *bound* to Vspec.

The choice **EmergencyPower** has a child **threshold:int**, which is another kind of VSpec called *variable*. As it provides decisions for choices, a resolution model provides *values* for variables. The "slot value assignment" variation point is bound to the variable **threshold**. This means that the value provided for this variable in the resolution model will be inserted into the attribute **threshold** in the SysML model.

What we have done up to now is define a variability model against a SysML base model, which together comprise a *product line model*. To derive a product from a product line a resolution model is supplied, which *resolves* the VSpecs in the variability model. Choices are resolved by deciding True or False. Variables are resolved by providing values. Other kinds of VSpecs not shown in this example have their own kinds of resolutions. Once a resolution model is provided, the product line model is transformed into a product model by applying the variation points as prescribed by the resolution model. This process of deriving a product model form a product line model given a resolution model is called *materialization*.

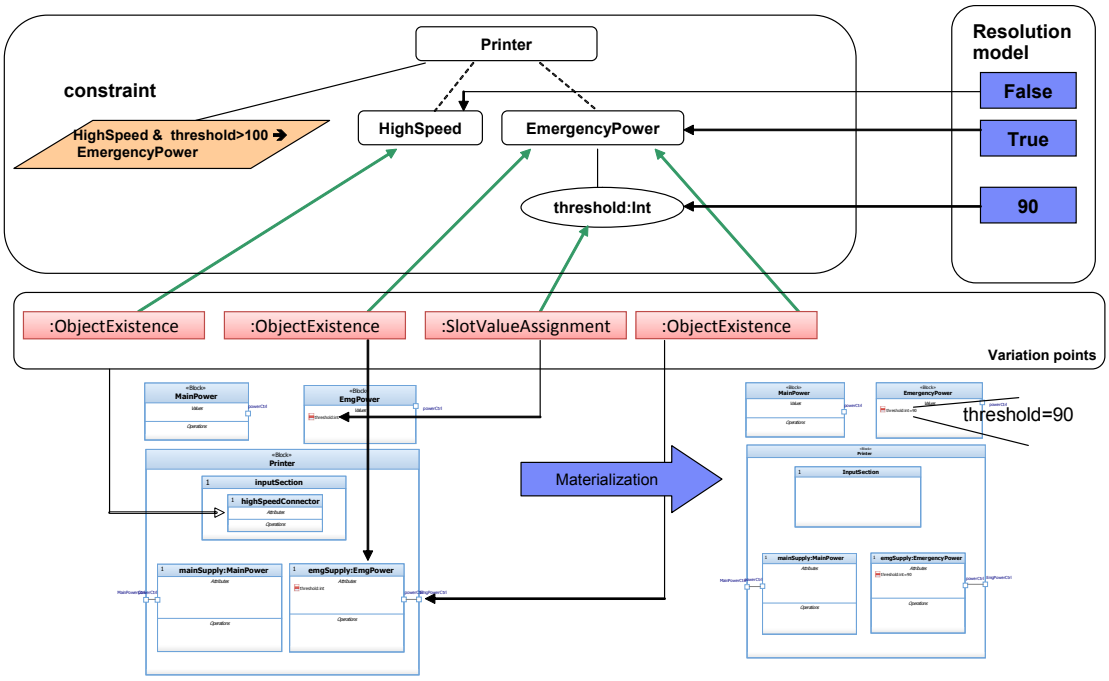



**Figure 4 – Variability model, Reolution model and Materialization**

Figure 4 shows the entire CVL cycle. On the right hand side, the resolution model decides **HighSpeed** negatively, **EmergencyPower** positively, and provides the value 90 for **threshold**. As a result the product line model is materialized into a product model as shown at the bottom. In the materialized model, the part

[1] For those familiar with feature models for defining product lines, VSpec trees are similar to feature models and deciding choices is similar to selecting features.

**highSpeedConnector** has been eliminated from the SysML design, the part **emgPower** and the port on **Printer** remain, and **threshold** has the value 90.

Figure 5 shows a materialization resulting from a different resolution model, this time with **HighSpeed**=True and **EmergencyPower**=False. Given the latter there is no need to provide a value from **threshold**. With this resolution model, the SysML part **highSpeedConnector** remains but the part **emgPower** as well as the port on **Printer** are eliminated. For less cluttering, in Figure 5 the variation points are depicted as in-model annotations. These annotations are for the sake of this figure only and are not part of the CVL concrete notation.

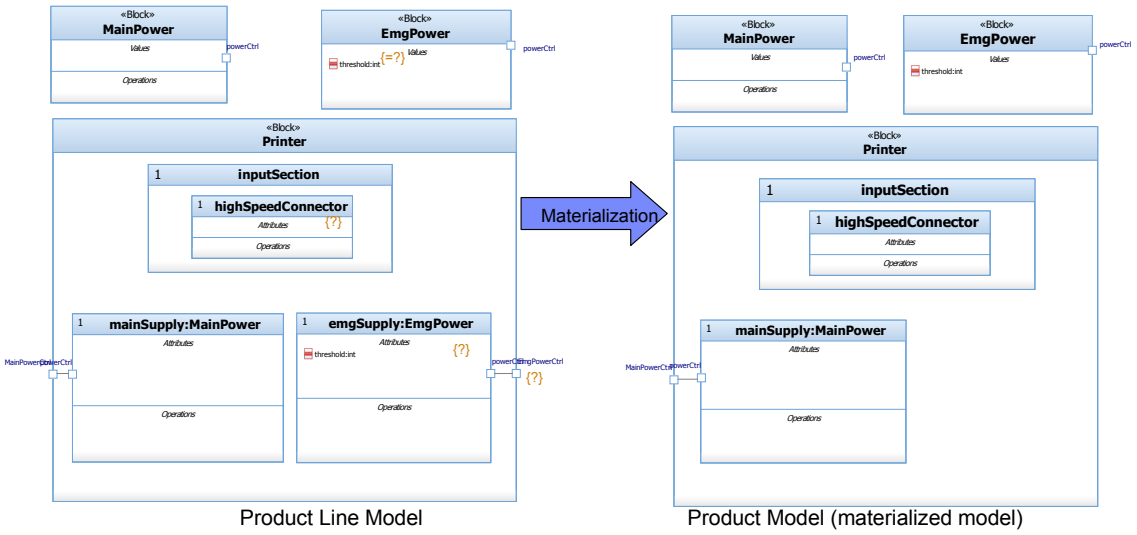



**Figure 5 - Materialization with HighSpeed=False and EmergencyPower=true**

Figure 6 shows the architecture of CVL. At the bottom, the *base model* is not expressed in CVL. The base model represents an instance of an arbitrary MOF metamodel, such as UML, on which variability is specified using CVL. From the standpoint of CVL, the base model is just a collection of objects and links between them. CVL itself has several parts as shown in the figure and described briefly below.

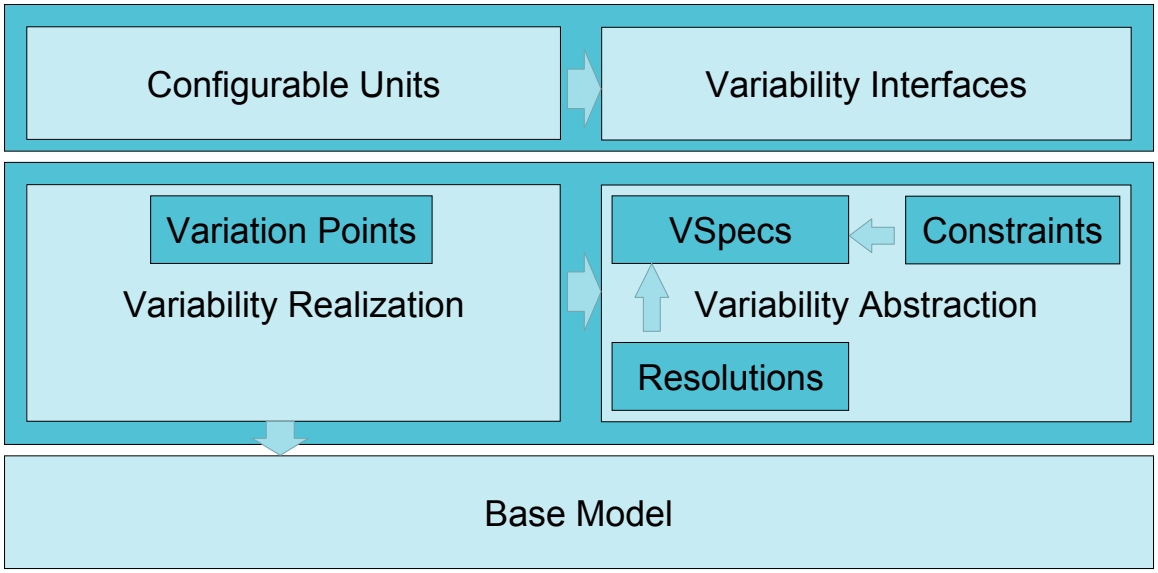



**Figure 6 - CVL architecture**

**Variability abstraction** provides constructs for specifying variability without defining the concrete consequences on the base model, hence *abstract*. It isolates the logical component of CVL from the parts

that manipulate the base model. The main variability abstraction construct is a variability specification (VSpec), which may represent a binary choice, a value parameter, or a specification element that can be instantiated several times. VSpecs can be arranged into trees.[2] *Constraints* can be associated with the nodes of VSpec trees to define restrictions on valid variability resolutions beyond those implied by the tree structure.

**Variability realization** provides constructs for specifying *variation points* on the base model, as shown in Figure 6. A variation point is a modification applied to the base model during the process of transforming the base model into a product model, called *materialization*. This is the part of CVL that impacts the base model. The variation points refer to base model elements via *base model handles*. The variation points also refer to the VSpecs to define what abstract notion of variability the variation point actually realizes. These references form the *binding* between the Variability Abstraction and the Variability Realization.

**Configurable units** exist on top of the previous constructs. They provide constructs facilitating the specification of configurable, reusable components. They also facilitate modularity by supporting compositional hierarchies, mirroring the compositional structure of the base model. Configurable units expose *variability interface,* made of VSpecs, through which they can be configured.

## 7.2 Variability Abstraction

### 7.2.1 VSpec Trees

Variability abstraction provides constructs for specifying and resolving variability on an abstract level, i.e. without specifying the exact nature of the variability w.r.t. the base model. The central concept is that of a *variability specification*, which we abbreviate as *VSpec*. VSpecs are technically similar to *features* in feature modeling, but we think of them as abstract variability specifiers. For example, instead of thinking of "GPS" as a feature that a camera might or might not have we think of it as a *choice*, which can be decided negatively or positively. There is, however, nothing to prevent implementers of CVL to use VSpecs for feature modeling.

A VSpec is an indication of variability in the base model. The specifics of the variability, i.e. what base model elements are involved and how they are affected, is not specified, which is what makes VSpecs abstract. In CVL, the effect on the base model is specified by *binding variation points* to VSpecs, which refer to the base model. Variation points and binding are discussed in Sec. 7.4.

VSpecs may be arranged as trees, where the parent-child relationship organizes the resolution space by imposing structure and logic on permissible resolutions.

### 7.2.2 Kinds of Vspecs

There are four kinds of VSpecs, three of which are discussed here: choices, variables, and variability classifiers. The fourth kind of VSpec – composite VSpec – is discussed in Sec.7.5.2.

Figure 7 shows a VSpec tree. The nodes shown as rectangles with rounded corners - **Office**, **Printer**, **Scanner**, **GreenMode**, **Type**, **Color**, **BW**, **Speed**, **High**, **Low**, and **Turbo** – represent choices.

A *choice* is a VSpec whose resolution requires a yes/no decision. Nothing is known about the nature of the choice in the level of a VSpec tree, except of course what is suggested by its name. For example, the fact that there is a choice **Scanner** in the tree indicates that in the resolution process we will need to decide yes or no about **Scanner**, and that this decision may have some effect on the base model, the nature of which is

[2] The concrete syntax of VSpec trees is similar to what are traditionally called "feature models"---tree-like menus of selectable features

unknown at the abstraction level. It could decide for instance whether or not a given element will be deleted, a given substitution will be performed, a link will be redirected, etc.

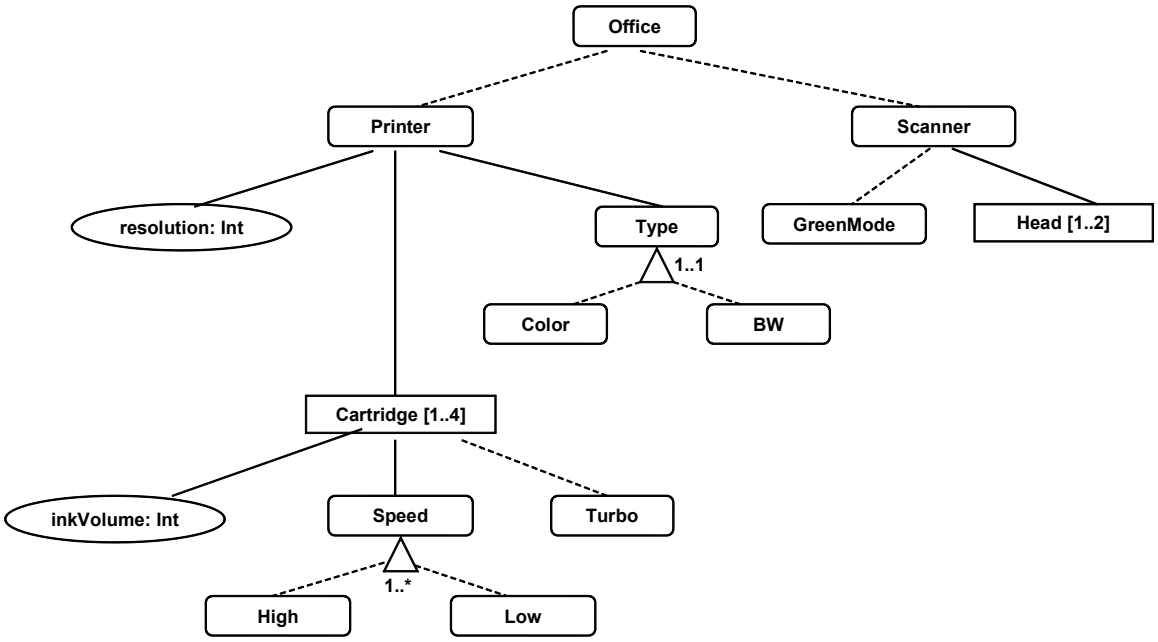



**Figure 7 - A VSpec tree**

A *variable* is a VSpec whose resolution involves providing a value of a specified type. This value is meant to be used in the base model, but similar to choices, it is unknown in this level exactly where and how. The variables in Figure 7 - **resolution** and **inkVolume** – are both of type **Int** and are shown as ellipses.

The third and more advanced kind of VSpec is a *variability classifier,* which we abbreviate as VClassifier. A VClassifier is a kind of VSpec whose resolution means creating instances and then providing per-instance resolutions for the VSpecs in its sub-tree. In Figure 7 there are two VClassifers – **Cartridge** and **Head** – shown as a regular rectangle. For each instance of Cartridge created as resolution we will need to resolve its **inkVolume**, **Speed**, **High**, **Low**, and **Turbo**. Thus, two different instance of Cartridge may have different resolutions for these VSpecs. Like for choices and variables, it is unknown at this level what the effect of each instance will be in the materialized product model.

Each VClassifier has an *instance multiplicity* which indicates how many instances of it may be created under its parent in permissible resolutions. This is elaborated more below. The multiplicity of **Cartridge** is [1..4], meaning between 1 and 4 instances must be created.

### 7.2.3 The Meaning of the Tree Structure

The sub-tree under a node represents "subordinate" VSpecs in the sense that the resolution of a node imposes certain constraints on the resolutions of the nodes in its sub-tree. In addition to the constraints implicitly imposed by the tree structure it is possible to specify explicit constraints, such as [**Scanner implies Color**]. The CVL constraints language is discussed in Sec. 7.3. Here we focus on the constraints expressible through the tree structure, which are made precise in Sec. 8.2.1 below.

- **Negative resolution implication** - A negative choice resolution implies a negative resolution for its sub-choices and no resolutions for all other child VSpecs. For example, if we decide "no" for **Printer** then we must decide "no" for **Type**, we cannot have any instance of **Cartridge**, and we cannot set a value for **resolution**, and so on recursively down the tree.

- **Positive resolution implication** – Each choice has a field "isImpliedByParent" which, when True, indicates that if its parent is resolved positively or the choice is a root then it must be decided positively. A resolution for a non-choice VSpec is always considered positive for this definition.

  The general rule is as follows: if a parent is resolved positively, i.e. it is either a positive choice decision or any variable resolution or any instance, then its sub-choices with isImpliedByParent =True must be resolved positively, its sub-variables must be given a value, and its sub-classifiers must be instantiated according to their instance multiplicity (see below). Further, a root choice with isImpliedByParent=True must be decided positively, a root variable must be given a value, and a root classifier must be instantiated according to its multiplicity.

  In Figure 7, a True value for "isImpliedByParent" is indicated by a solid edge from the parent, and a False value is indicated by a dashed edge. For non-choice VSpecs, the line is always solid. *Notation-wise*, for a root choice the "isImpliedByParent" field is assumed to be True unless it is also a leaf, i.e. an isolated node, in which case it is assumed to be False. This way, isolated choices can be decided positively or negatively, whereas root choices with children must be decided positively.

  According to the rules above, **Office**, **Type**, and **Speed** have isImpliedByParent=True and the other choices have isImpliedByParent=False. If we decide True for **Printer** we must decide True for **Type**, we must provide an Integer value for **resolution**, and we must create between 1 and 4 instances of **Cartridge** (see below for instance multiplicity), where for each cartridge we must decide True for **Speed** and provide an Integer value for **inkVolume**. Similarly, if we decide True for **Scanner** then we must create between 1 and 2 instances of **Head**.

- **Group Multiplicity** – A VSpec may (optionally) have *a group multiplicity,* specifying how many total positive resolutions must be under it in case it is resolved positively. *Positive resolution* means the same as above, i.e. a positive choice decision or any variable value assignment or any instance of a VClassifier. In Figure 7 group multiplicities are shown using small triangles: if **Type** is decided positively then exactly one of **BW** and **Color** must be decided positively and the other negatively. Similarly, for each instance of Cartridge, **Speed** must be decided positively and then either one of or both **Low** and **High** must be decided positively.

- **Instance Multiplicity –** Each Classifier has an instance multiplicity specifying how many instances of it must be created. This is specified as a range using a lower multiplicity and an upper multiplicity.

### 7.2.4 VSpec Derivations

CVL supports a concept of *VSpec derivations*, specifying that the resolution of some given VSpec is to be derived, i.e. calculated, from the resolution of other VSpecs. This is similar to UML derived properties. A resolution model does not need to provide resolutions for derived VSpecs since they are derived from other resolutions. This is in contrast with constraints, which serve to verify that resolutions satisfy certain restrictions.

While each variation point must be bound to a single VSpec, VSpec derivations make it possible to effectively map a variation point to an arbitrary logical condition or arithmetic expression by binding the variation point to a VSpec derived though a formula or expression.

A derived choice is specified using a formula over other VSpecs which evaluates to True/False. In CVL formulas are called constraints, so the form of a choice derivation is $c \leftarrow Cons$, where *c* is the derived choice and *Cons* is a constraint whose evaluated truth value is used to determine the value of *c*. For example, we could specify **Printer** as a derived choice as follows: **Printer ← Scanner and GreenMode**. This means that the resolution for **Printer** is calculated as the logical conjunction of the resolutions for **Scanner** and **GreenMode**: thus, a resolution model does not need to specify a value for **Printer**. Note that

the "←" here represents derivation, not logical implication, so **Printer** is assigned exactly the truth value of the constraint [**Scanner and GreenMode**].

Similarly, a variable derivation has the form *v* ← *Exp* where *v* is a variable and *Exp* is an expression in the CVL constraint language, which evaluates to a value. For example we specify that the variable resolution is derived as follows: **resolution ← minRes*2**, where **minRes** is some other variable (not shown in Figure 7).

Classifiers cannot be derived. The forth kind of VSpec – composite VSpecs – may be derived. This is discussed in Section 7.5.5.

### 7.2.5 VSpec Resolution Trees

VSpecs are resolved by *VSpec resolutions*. There are four kinds of VSpec resolutions which mirror the four kinds of VSpecs, three of which are described here, and the fourth– variability configuration (VConfiguration) – in Sec. 7.5.2. *Choice resolutions* resolve choices; *variable value assignments* resolve variables; and *VInstances* resolve VClassifiers. Each VSpec resolution resolves exactly one VSpec of the appropriate kind. In the absence of classifiers each VSpec is resolved by at most one VSpec resolution. For a VSpec tree with VClassifiers, a single VSpec may have several VSpec resolutions resolving it. No ambiguity is created however due to the resolution tree structure which we discuss next.

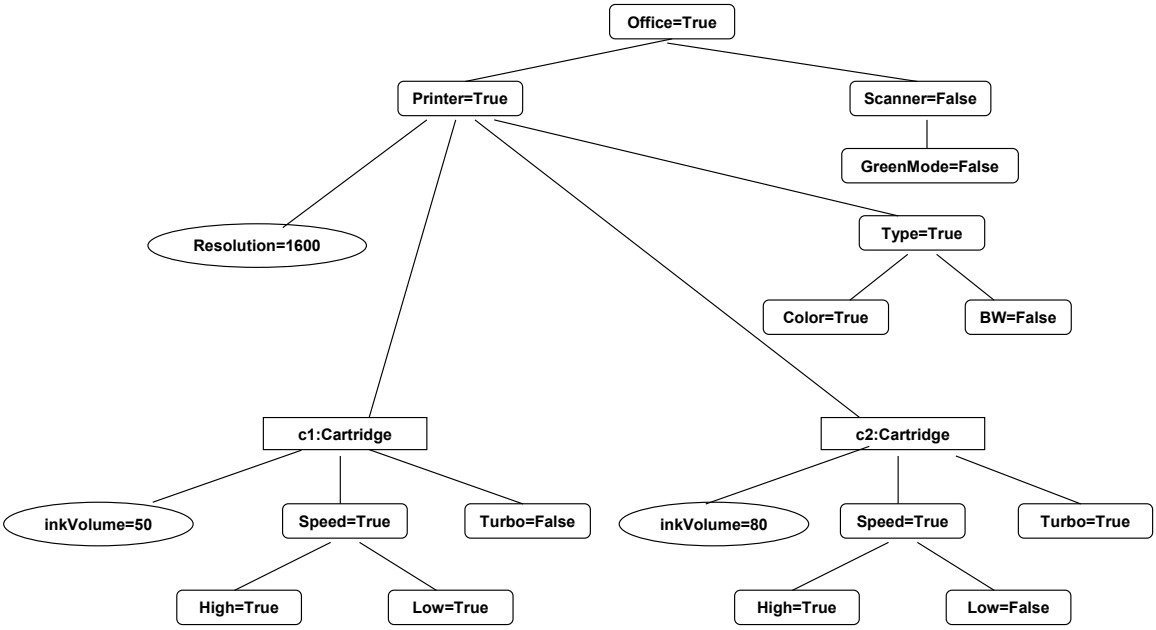



**Figure 8 - A VSpec Resolution tree**

In Figure 8, a sample VSpec resolution tree corresponding to the VSpec tree of Fig. Figure 7 is shown. Each node in the tree is a VSpec resolution and references exactly one VSpec. To avoid cluttering, these references are specified using the names of the resolved VSpecs rather than arrows. This is just an auxiliary notation – a CVL model would contain references to resolved VSpecs. For example, in Figure 8, **Printer=True** is a choice resolution resolving **Printer** positively; **resolution**=1600 is a variable value assignment resolving **resolution**; and **c1:Cartridge** as well as **c2:Cartridge** are instances, i.e. resolutions of, **Cartridge**. Note that **Low=True** on the left side and **Low=False** on the right side are two different choice resolutions (i.e. different CVL elements) resolving the same choice **Low**.

The tree structure of VSpec resolutions mirrors that of the VSpecs they resolve; thus, whenever one VSpec is a child of another, each resolution for the child will be a child of a resolution for the parent. The VSpec resolutions must respect the rules outlined in Sec. 7.2.3 above. Consequently, in Figure 8, **Printer** is decided positively, so is **Type**, **resolution** is assigned a value (1600), and there are 2 two instances of

**Cartridge**, **c1** and **c2**, which is between 1 and 4. Since **Scanner** is decided negatively so is **GreenMode**, and there are no instances of **Head** despite the fact that its instance multiplicity is 1..2.

## 7.3 Constraints

In CVL, constraints are used to express intricate relationships among VSpecs that cannot be directly captured by hierarchical relations in a VSpec tree. To this end CVL introduces a basic constraint language, a restricted subset of The Object Constraint Language (OCL), fully defined in this specification. In addition CVL allows the use of other constraint languages, including more complete OCL.

The following subsections illustrate the use of both the restricted subset and the complete OCL for expressing constraints over VSpecs using examples. Sections 7.3.1 through 7.3.5 cover the basic constraint language; the remaining subsections cover the use of the complete OCL. Further, Sections 7.3.1 through 7.3.4 cover the subset of the basic constraint language that does not deal with classifiers; readers not interested in classifiers may skip the remaining subsections. Section 7.3.5 illustrates the rudimentary support for classifiers provided by the basic constraint language. The definition of the basic constraint language can be found in Section 8. The application of the complete OCL to CVL is described in Appendix 16.2.

### 7.3.1 Propositional Constraints (Basic Constraint Language)

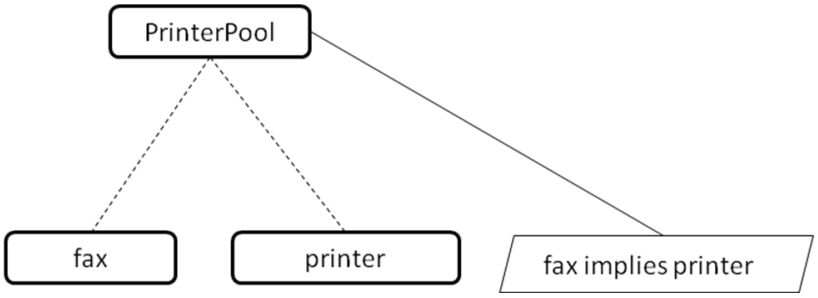



**Figure 9 – Simple propositional constraint**

Figure 9 shows a variability specification of a pool of printers that provide a fax service and a printing service (printer). Both specification choices fax and printer are optional; however, printer is required for delivering the fax functionality in order to be able to print incoming faxes. Therefore, fax requires the presence of printer. This dependency is expressed by a propositional constraint (in the parallelogram), where the implication (‘implies’) stands for 'requires'. If there is a fax, then the printer must be present. Otherwise, there may or may not be any printer in the system. The solid line between the root choice and the parallelogram specifies PrinterPool as the context for the constraint.

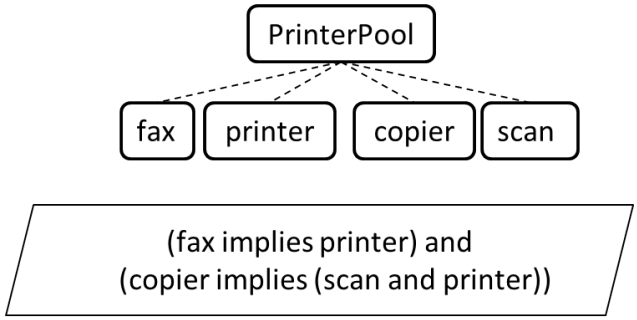



**Figure 10 – Complex propositional constraint**

Figure 10 shows a variability specification of a printer pool that offers optional choices of fax, printer, scanner, and copier. As previously, the printer provides part of the fax functionality, as it prints the incoming documents. Furthermore, the printer pool can copy documents if both a printer and a scanner are

present. The two constraints are shown in the parallelogram, combined using conjunction (and). The first implication is the same as in Figure 9 (fax implies printer). The second implication specifies that the copier functionality requires both printer and scanner to be present. Since there is no line connecting the constraint to any VSpec, the constraint is global (within its containing variability package or configurable unit). In this specific example, specifying the constraint in the context of PrinterPool would have the same effect as specifying it as global.

### 7.3.2 Arithmetic Constraints (Basic Constraint Language)

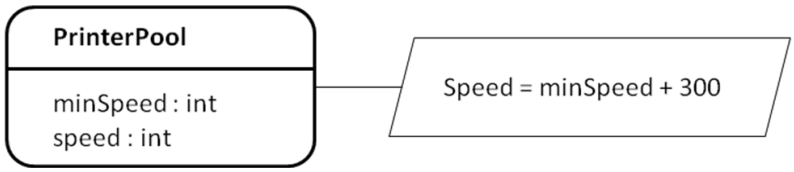



**Figure 11 – Arithmetic propositional constraint**

CVL constraints may involve expressions to restrict values of variables. Figure 11 shows variability specification of a printer pool that has two variables: minimal speed (minSpeed) and the normal printing speed (speed). Note that the figure uses the abbreviated notation for variables by listing them within their parent VSpec rather than showing them as separate oval nodes. The relationship between minSpeed and speed is defined by a constraint: speed is equal to minSpeed enlarged by a constant.

### 7.3.3 Path Expressions (Basic Constraint Language)

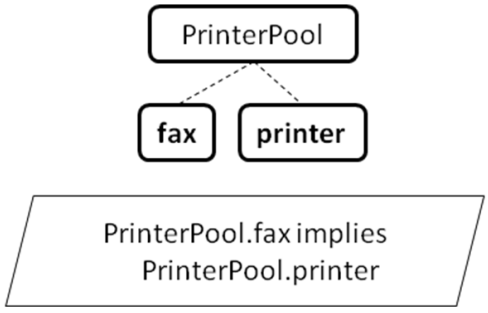



**Figure 12 – Path expression**

Figure 12 shows variability specification similar to Figure 9. This time the very same constraint as before, that fax requires printer, is expressed using fully qualified names; path expressions that navigate along the VSpec hierarchy. Path expressions are useful to disambiguate VSpec names if multiple VSpecs use the same name. For example, if both fax and printer had a child choice named color, we could distinguish them as fax.color and printer.color.

### 7.3.4 Context of Constraints (Basic Constraint Language)

We have already seen that constraints can be written in context (see for instance in Figure 9). Indeed, many constraints are easier to comprehend if they are written in the context of the constrained element. Figure 13 has two constraints, one in the context of PrinterPool and another one in the context of fax. Since the latter one refers only to the children of fax, it is naturally placed there. In this example, the constraint on fax could also be attached to PrinterPool without changing the meaning of this model; however, if printer had a child choice also named color, moving the constraint from fax to PrinterPool would require using an explicit path to refer to color, i.e., fax.color, in order to avoid ambiguity. Attaching the constraint to fax avoids such ambiguities, as context is used as the starting point for name lookup.

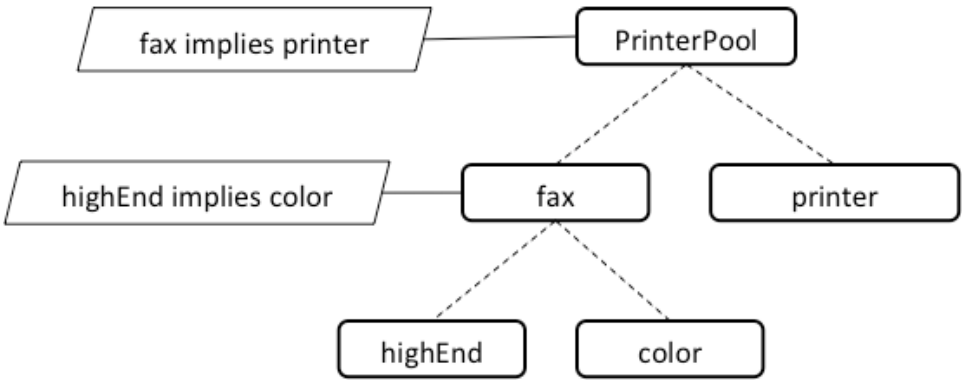



**Figure 13 Constraints in context**

In general, however, context influences the semantics of a constraint: a constraint attached to a VSpec is enforced only if the VSpec is resolved positively. That is, a constraint attached to a choice is enforced only if the choice is decided positively. Similarly, a constraint attached to a variable is enforced if the parent of the variable is positively resolved, in which case the variable must have a resolution (i.e., value assignment), too. For classifiers, the constraint is enforced for every instance of the classifier, as will be explained in the next section.

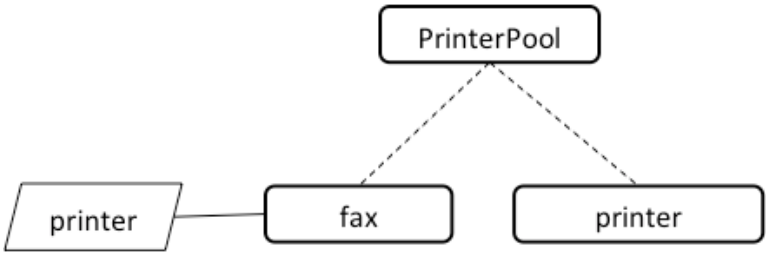



**Figure 14 Constraint in context capturing a condition (equivalent to Figure 9)**

Figure 14 shows how to use context in order to express conditions on choices. By attaching the constraint *printer* to fax, printer has to be positively decided whenever fax is positively decided. The condition *printer* on fax has equivalent meaning to attaching *fax implies printer* to PrinterPool, as in Figure 9.

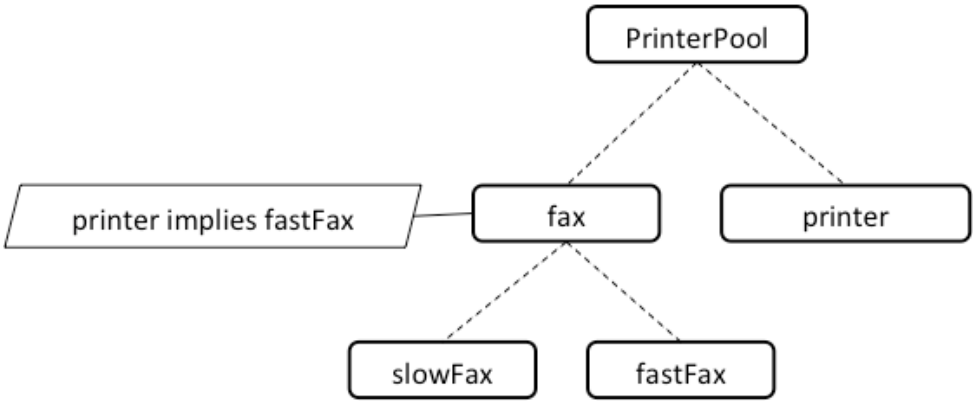



**Figure 15 Constraint simplification using context**

Figure 15 shows that stating a constraint in a context as opposed to globally can simplify the constraint. The constraint attached to fax could also be written as the following global constraint: *fax and printer implies fastFax*. By stating the constraint in the context of fax, we only need to state that *printer implies fastFax* since this constraint is enforced only if fax is decided positively. Note that the examples in Figure 14 and Figure 15 rely on the ability of a constraint to refer to VSpecs outside its context, such as printer in these two examples.

### 7.3.5 Implicit Quantification (Basic Constraint Language)

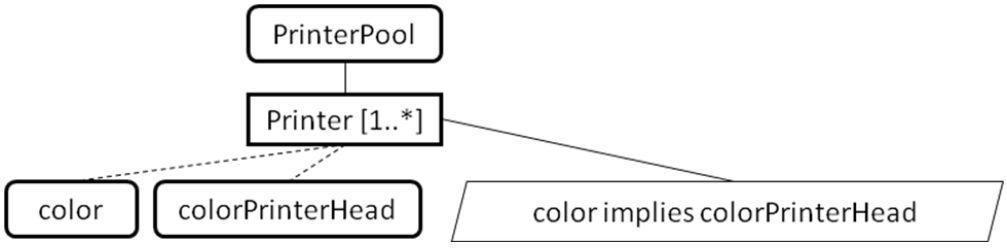



**Figure 16 – Implicit quantification**

Figure 16 shows a variability specification of a printer pool containing multiple printers. In this example, a printer may be color and its head is also optionally color. If a printer is to print color documents, then it must have a color head. This intention is expressed as a propositional constraint in the parallelogram. The propositional constraint is specified in the context of the Printer VClassifier. Although the constraint concerns a VClassifier, there is no explicit quantifier. It is implicit that the constraint holds for all instances of Printer.

### 7.3.6 Quantification (Complete OCL)

Propositional formulas are suitable for specifying constraints over choices, but they are insufficiently expressive to constrain VClassifiers. Unlike choices, which are binary yes/no decisions, VClassifiers are sets of elements. Quantification over sets is achieved by using existential (∃) or universal (∀) quantifiers. CVL uses Object Constraint Language (OCL) to provide syntax and semantics for quantifiers.

### 7.3.7 Existential Quantification (Complete OCL)

Existential quantification specifies that in a given set there is at least one element satisfying accompanying predicate.

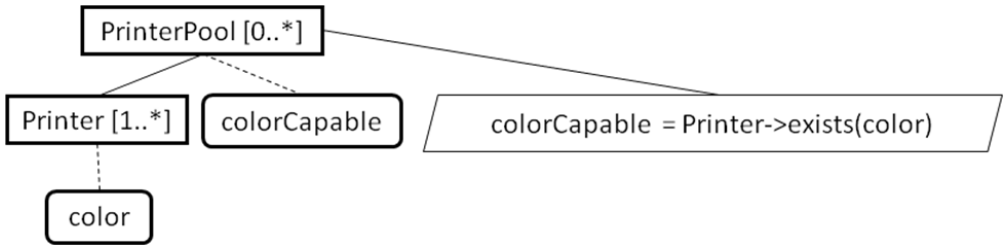



**Figure 17 – Existential quantification**

Figure 17 shows variability model of a printer pool. In the model there may be any number of pools, and each pool has at least one Printer. The Printer is either color or black-and-white. The colorCapable choice is selected *if and only if* there exists at least one color printer in the printer pool. To capture this intention, we specify a dependency in the parallelogram node.

The constraint uses equality operator (=) for logical equivalence. The Printer VClassifier specifies a set of elements, therefore it is an OCL collection. In OCL, quantifiers (i.e. ∃ and ∀) act on collections that they follow. Syntax of OCL quantifiers resembles a function call. In the example, the constraint requires that within each color capable printer pool there exists a printer with the color choice present.

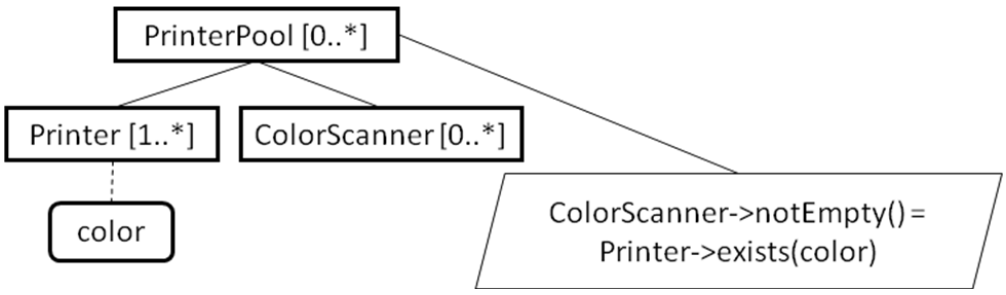



**Figure 18 – Existential quantification over two VClassifiers**

Dependencies are not limited to choices and VClassifiers, but can occur among two or more VClassifiers. In the next example (Figure 18), PrinterPool has at least one Printer and any number of ColorScanners. Some printers may print in color, but color scanners have no further options. There is an additional dependency between Printer and ColorScanner: scanning color images makes sense if an only if there is a printer that can print images in color.

Both ColorScanner and Printer are OCL collections. The constraint says that within each printer pool there is at least one ColorScanner (the collection is not empty) whenever there is a color printer. Note that when an element is a choice then the choice is evaluated to true or false, such as colorCapable. In case of VClassifiers, the function nonEmpty() is used to state that there is at least one instance of the VClassifier.

### 7.3.8 Universal Quantification (Complete OCL)

Universal quantification requires that all elements of a set have certain property.

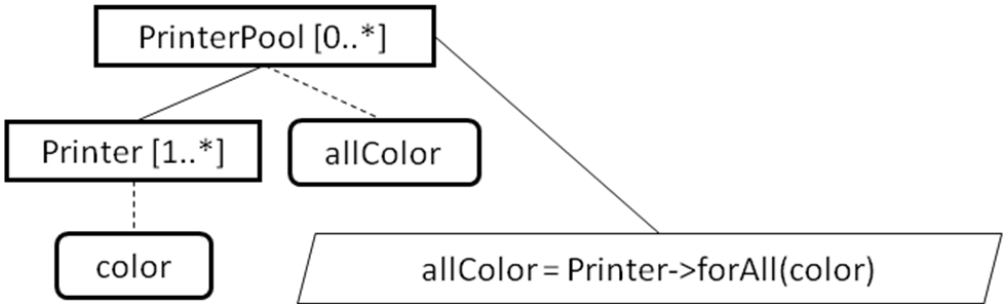



**Figure 19 – Universal quantification**

Figure 19 shows another variant of the printer pool example. Selecting allColor is made equivalent to requiring that all printers in a pool are color. It is expressed by attaching a constraint to PrinterPool: allColor is selected whenever all Printers print in color.

Similarly to existential quantification, the forAll quantifier follows the Printer collection. Its argument (color) must be present in all printer instances.

### 7.3.9 Context and Universal Quantification (Complete OCL)

We have already seen that context enables implicit quantification (see for instance in Figure 16). We can use this capability to simplify constraints that use explicit universal quantification. In Figure 20 each PrinterPool has an optional Fax. Printers and fax are either color or black-and-white. We require however, that color Fax is only available if all printers are color. The same dependency can be specified as a constraint under different elements (compare Figure 20 and Figure 21).

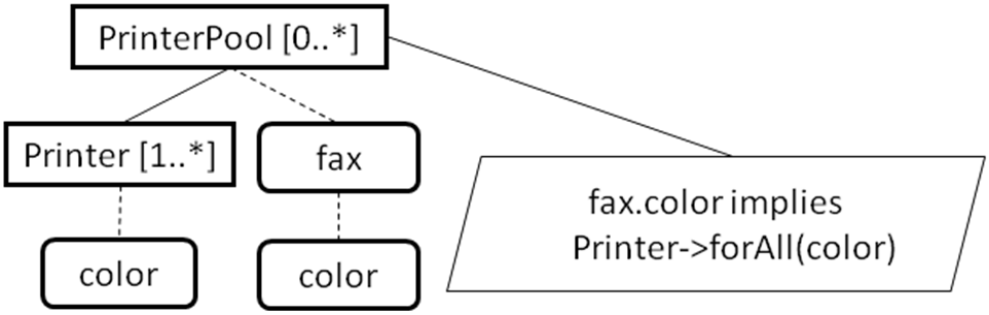



**Figure 20 – Constraint without local context**

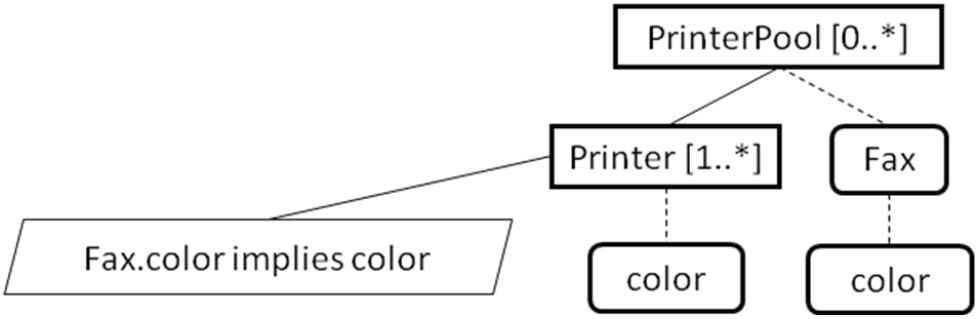



**Figure 21 – Constraint in local context**

In the first case (Figure 20) universal quantification over all printers is used to enforce that they can print in color. Figure 21 expresses the same intention, but instead of explicitly quantifying over all printers, it places a constraint in the context of Printer. Such a constraint will be automatically enforced for all instances of Printer. It states that for each Printer the color Fax functionality requires the Printer to be color. In this example, Fax.color is unambiguously resolved as a sibling of the context (Printer) in the VSpec tree. Detailed name resolution rules are given in Section 8.3.5.

### 7.3.10 Constraints over Sets (Complete OCL)

So far we explored propositional formulas with quantifiers. The existential quantifier states that there is at least one element with certain property. When necessary to be more precise about the number of instances for which a property, one can use cardinality constraints.

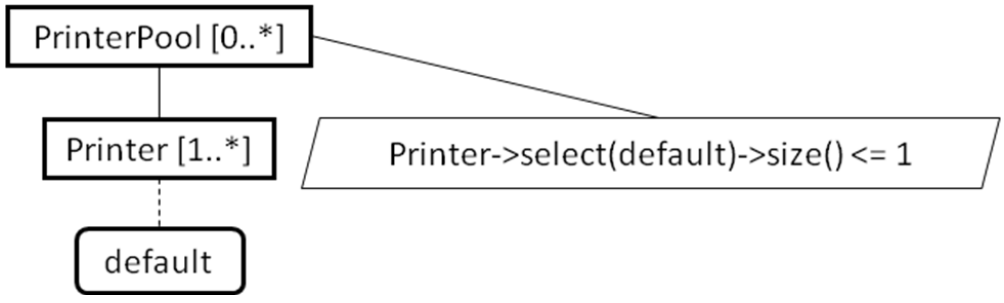



**Figure 22 – Cardinality constraint over a set**

Figure 22 demonstrates such a constraint. As previously, there is at least one Printer in a pool. However, one printer can be selected as the default device, determined by the choice default under Printer. The constraint states that there is at most one default printer in any given pool. Cardinality constraints count the number of set elements and compare with another natural number.

The constraint is composed of several parts. First, Printer → select(default) returns a collection of those printers that have the default choice selected. Then, the size() operator is applied to count the number of elements in the collection. Finally, the resulting number is compared with one.

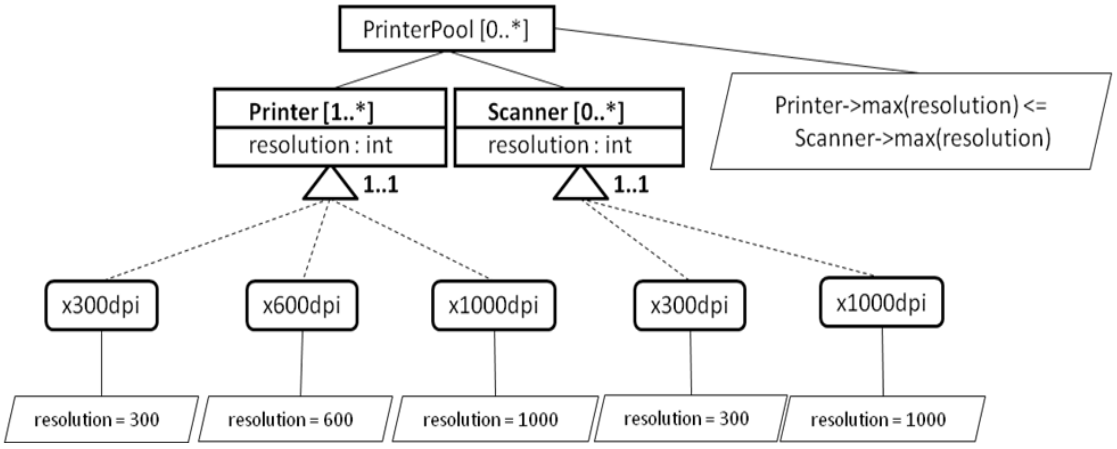



**Figure 23 – Arithmetic constraints with VClassifier**

Arithmetic expressions can also be combined with collection style operations of OCL. Figure 23 shows PrinterPool with Printers and Scanners. Both VClassifiers have variables (resolution) representing device resolution. The variables store values of type int, i.e., integers. Printer's resolution is controlled by three choices, of which only one can be selected (indicated by the group cardinality 1..1). Each choice has a constraint written in its context saying that resolution (which resolves to the resolution variable in Printer) must be equal to given number, e.g., if x300dpi is present, then resolution's value will be equal to 300. An analogous principle is applied to the Scanner VClassifier.
The top-most constraint states that in the entire PrinterPool's maximum printer resolution cannot be higher than maximum scanner resolution.
For both VClassifiers in the constraint, we apply function max() that takes a variable as a parameter. As Printer and Scanner are collections, max() returns the maximum resolution from each collection.

## 7.4 Variability Realization

### 7.4.1 Introduction to the Variability Realization

The variability realization mechanisms make it possible to *materialize* the products from the CVL description by transforming a base model in some MOF defined language to a product model in that same language. The realization defines *variation points* that represent specific simple modifications of the base model materializing it eventually into the product model. The generically defined mechanisms for variability realization assume that the base model is also a MOF model.

There are four ways that variation points define modifications of the base model:

- *Existence* – an indication that the existence of a particular object, link, or value in the base model is in question.
- *Substitution* – an indication that a single object or an entire model fragment may be substituted for another. *Object substitution* involves two objects and means redirecting all links in which one is involved to the other and then deleting the former. *Link-end substitution* involves a link-end and an object and means redirecting the link-end to the object (hence substituting it for the former object at this end). *Fragment substitution* involves identifying a *placement fragment* in the base model via *boundary elements*, thereby creating a conceptual "hole" to be filled by a *replacement fragment* of a compatible type.
- *Value assignment* – an indication that a value may be assigned to a particular slot of some base model object.

- *Opaque variation point* – an indication that a domain specific (user defined) variability is associated with the object(s), where the semantic of domain specific variability is specified explicitly using a suitable transformation language, such as QVT.

Each variation point references a VSpec, which in CVL is called *binding*: Binding provides the linkage between variability realization and variability abstraction. Each variation point is bound to a single VSpec. VSpecs are discussed in Sec. 7.2.1 above.

That a variation point is bound to a VSpec means the application of the variation point to the base model during materialization depends on the resolution for the VSpec. The nature of the dependency is specific to the kind of variation point. For *choice variation points*, which may be bound to a choice, the decision for the choice (True/False) determines whether or not they will be applied. For *parametric variation points*, which must be bound to a variable, the value for the variable provides the argument value. *Repeatable variation points* are applied once for every instance of the VClassifier to which they may be bound.

The above categorization of variation points into choice, parametric, and repeatable variation points is according to the nature of their binding to VSpecs. This categorization is in general orthogonal to the other, previous categorization into existences, substitution, value assignment, and opaque variation points which is according to the way they modify the base model. The taxonomy in the metamodel, shown in Figure 24, follows the choice/parametric/repeatable categorization. Note that fragment substitution is both a choice and a parametric variation point, so it *may* be bound to a choice or to a VClassifier but it *must* ultimately be bound to a single VSpec.

In this Chapter we will present three examples. The first example (Section 7.4.2) shows the variability of the Scanner where we apply a ChoiceVariationPoint and a ParametricVariationPoint. Our second example (Section 7.4.3), the PrinterPool, will show Fragment Substitution which is the most general of the defined variation points and that can correspond to VClassifier as well as Choice as it is a subclass of both RepeatableVariationPoint and ChoiceVariationPoint. The third example (Section 7.4.4) shows a scenario of Opaque Variation Point.

CompositeVariationPoints (that are bound to CVSpecs) will be explained with the Configurable Units (Section 7.5).

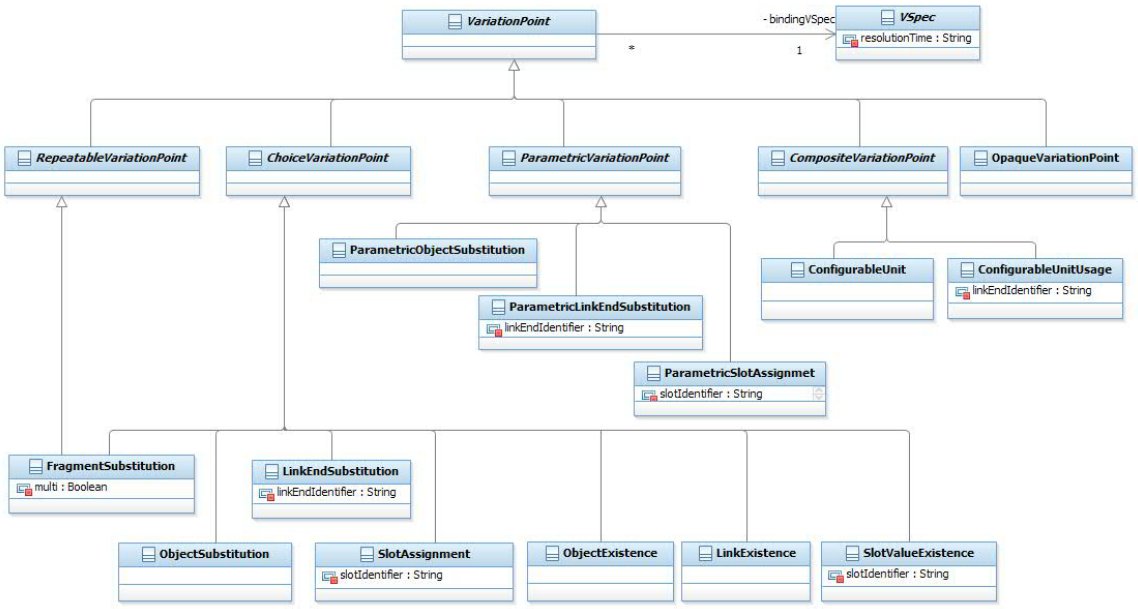



**Figure 24 The VariationPoint taxonomy**

### 7.4.2 Choice Variation Points and Parametric Variation Points

In this Section we will give examples where ChoiceVariationPoints bound to Choices, and ParametricVariationPoints bound to Variables are used. In Figure 25 we show the base model of the *Scanner*. The base model is given in UML using class definitions of *Feeder*, *Turner*, *Quality*, *BW*, *Color* and *Scanner*. The Scanner is also described by a composite structure where parts *f,q* and *t* are connected by connectors.

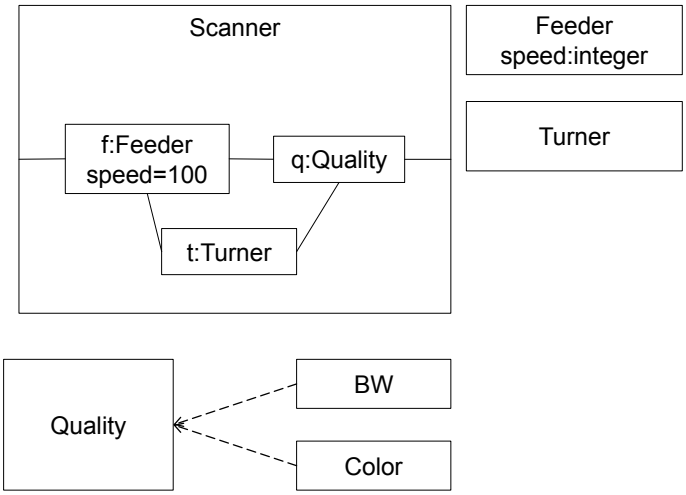



**Figure 25 The Scanner base model**

The scanner contains one part that feeds the paper to be scanned, a part that may turn the paper for scanning on both sides and finally an optical unit that determines the quality of the scanning. There are dependencies from classes BW and Color to Quality indicating that BW and Color can serve as Quality. These dependencies are just for illustrative purposes, no linkage between Color/BW and Quality is needed in the base model since this is expressed by the (external) CVL variability model. The feeder of the scanner is characterized by the speed that it can feed the paper through.

Our VSpec model of the Scanner is defined in Figure 26.

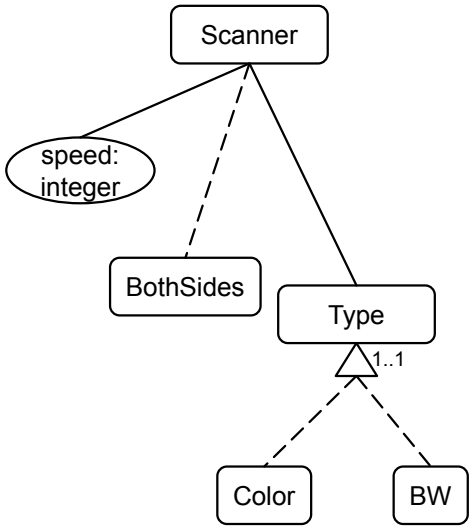



**Figure 26 The Scanner VSpec model**

The variability of the Scanner model says that the feeding speed is given by an integer value and that it is optional to have scanning of both sides. Furthermore, the scanner may be either a color scanner or a black and white scanner, but not both.

The realization model relates the VSpec model with the base model representing the realization of the individual choices to be made in the resolution model.

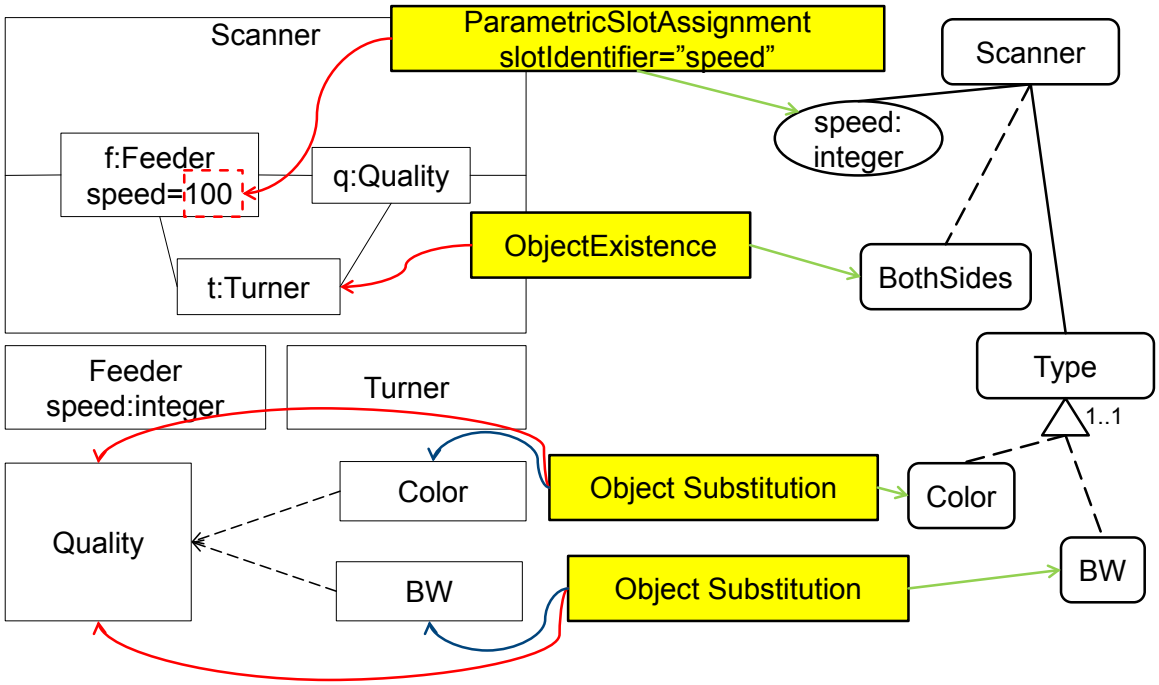



**Figure 27 Realization model of the scanner variability**

We have illustrated three different kinds of variation points in Figure 27.

The *ParametricSlotAssignment* is bound to the *speed* variable in the VSpec model. It takes its value from *speed*'s resolution and assigns it to the feeder speed in the base model.

The *ObjectExistence* is bound to the choice *BothSides*. It describes that when this choice is true, then the *t:Turner* should be present in the product model, while when the choice *BothSides* is false the *t:Turner* is not needed and shall be removed from the product model.

The *Type* choice defines that the scanner is either *Color* or *BW* (black & white). When *Color* is chosen this is done by the *ObjectSubstitution* bound to *Color*, where the object defining the class *Quality* is replaced by the class *Color* in the base model. The effect of the indicated substitution is that any property of type *Quality* in the original base model will be of type *Color* in such a product model.

When *BW* is chosen, in the same way, the corresponding *ObjectSubstitution*, bound to *BW* and referencing the class *BW,* will replace class *Quality* by class *BW*.

One possible product materialization and the corresponding product model are shown in Figure 28.

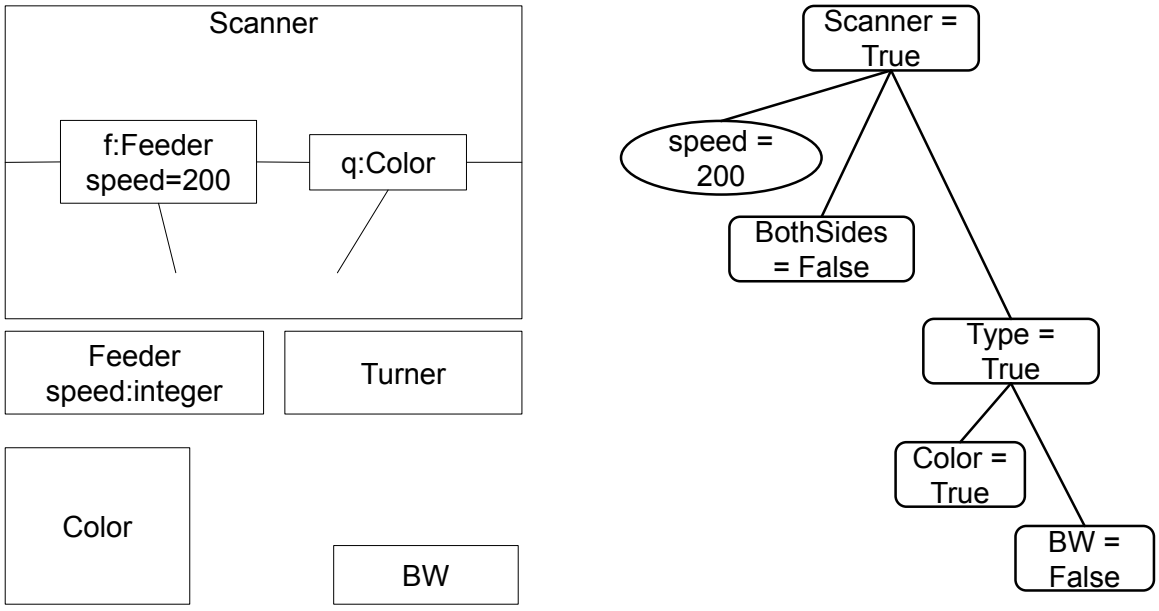



**Figure 28 Scanner resolution and product models**

Notice that our *Scanner* contains a couple of dangling connectors towards the non-existing *t:Turner*. This is to indicate that it is specific to the tool and/or the base language what cascading effects the removal of a given object shall be, hence the *possible* materialization above. Most UML tools would likely remove these dangling associations.

In some cases it is required to "map" a variation point to a logical formula over VSpecs. For instance, suppose in the example above we had inside the class *Scanner* an optional part *extraPower* which exists only if the choices *BothSides* and *Color* are both decided positively. To accommodate this, we would first define an *ObjectExistence* variation point referencing *extraPower*. But because each variation point in CVL is bound to a single VSpec we cannot directly "map" the variation point to the conjunction. Instead, we would introduce a new choice *extra*, bind the *ObjectExistence* to it, and derive it as ***BothSides and Color***. VSpec derivations are discussed in Sec. 7.2.4.

This Section has illustrated three different kinds of Variation Points. The ParametricSlotAssignment is one of three ParametricVariationPoints. The other two are ParametricLinkEndSubstitution and ParametricObjectSubstitution. ParametricLinkEndSubstitution changes a link such that the target object is provided by the resolution model. Similarly, ParametricObjectSubstitution corresponds to ObjectSubstitution with the replacement being provided by the resolution model.

SlotAssignment is changing the value of the slot into another specified value, and LinkEndSubstitution is changing the object to which the link is pointing.

SlotValueExistence and LinkExistence are the slot value and link equivalents of ObjectExistence.

### 7.4.3 Repeatable Variation Points

This Section introduces Fragment Substitution in the context of VClassifiers. Fragment Substitutions are the most general form of substitution. A sequence of fragment substitutions can express any change of the base model into any target. FragmentSubstitution works well to realize VClassifiers, but are not limited to being applied for VClassifiers.

A Fragment Substitution is a repeatable variation point when the *multi* property is set to *true* and referring to a VClassifier. A repeatable variation point is applied repeatedly and can be used to produce new and

more elements in the base model. The resolution model decides how many repetitions will be performed. For every time the repeatable variation point is applied, another copy of a replacement fragment is cumulated onto the placement. Thus this gives the opportunity to derive additive variability where the base model is being augmented.

Every new instantiation of a VClassifier defines its own scope and every variation point referring a VSpec contained in the VClassifier will apply to that scope. In our example the color and speed choices are set separately for two printers in the resolution model as will be shown in Figure 31.

We shall now show the details of such *RepeatableVariationPoints* in an example with a *PrinterPool* base model as shown in Figure 29.

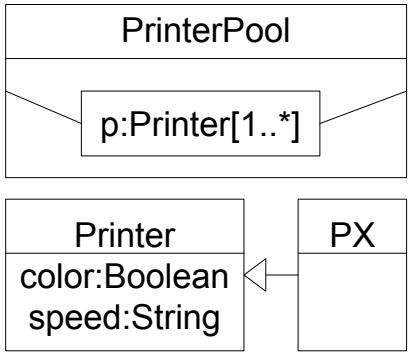



**Figure 29 The PrinterPool base model**

The base model shows a *PrinterPool* containing a set p of *Printers*. The *Printer* is subclassed to *PX* where e.g. the default color and speed values can be given. When we want a given small set of different printers, we can define a number of subclasses to *Printer* similar to *PX*, but individually different where each one of them represents the refined definition of a *Printer*.

The VSpec tree of *PrinterPool* with VClassifier *Printer* is shown in Figure 30.

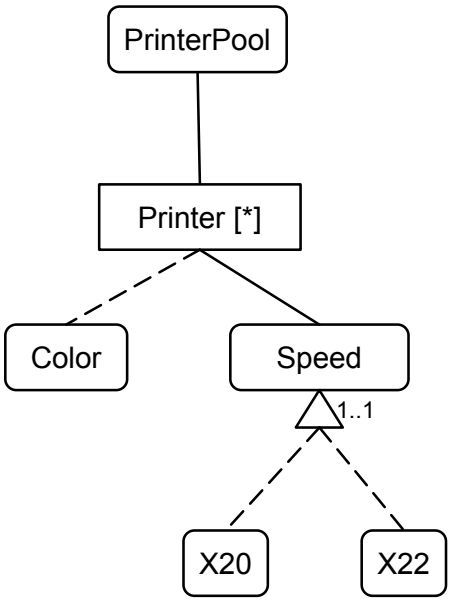



**Figure 30 VSpec tree of the PrinterPool**

The printer pool consists of a number of printers, and for each of these printers we decide whether it has colors and what speed it can support (either *x20* or *x22*).

One specific resolution with two printers is given in Figure 31 where one printer has color and supports *x20* and the other has no color and supports *x22*.

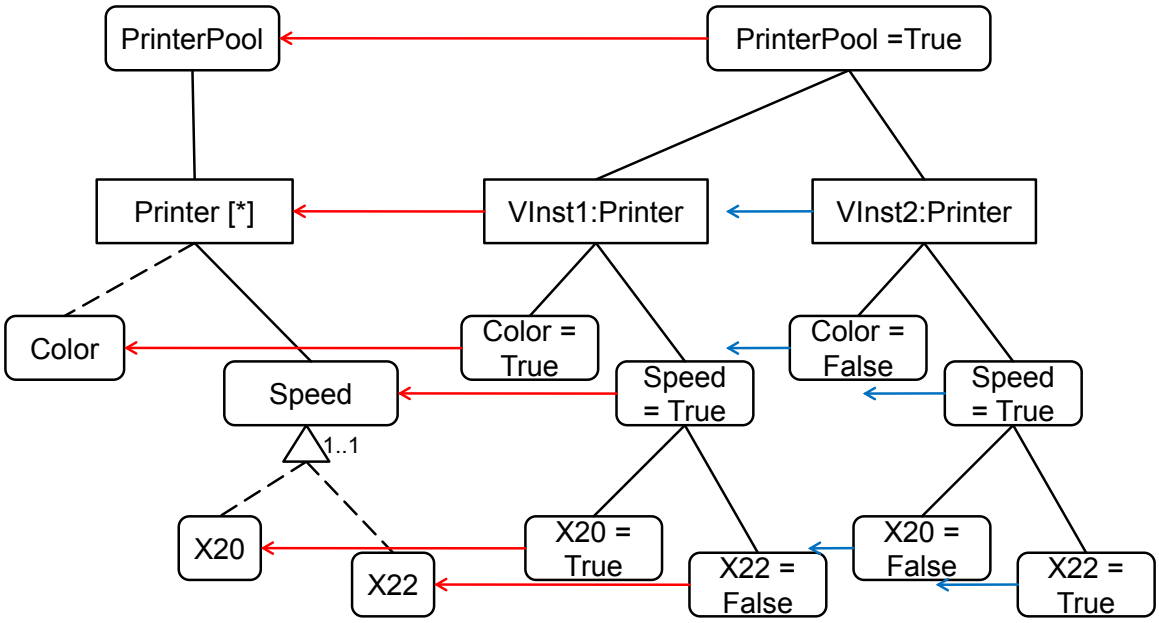



**Figure 31 VSpec and Resolution models of one particular printer pool**

We indicate in Figure 31 by red and blue arrows the relationships between the resolution elements and the VSpecs. The blue arrows have not been drawn all the way, but the VSpecResolution identifiers clearly show where the correspondence is.

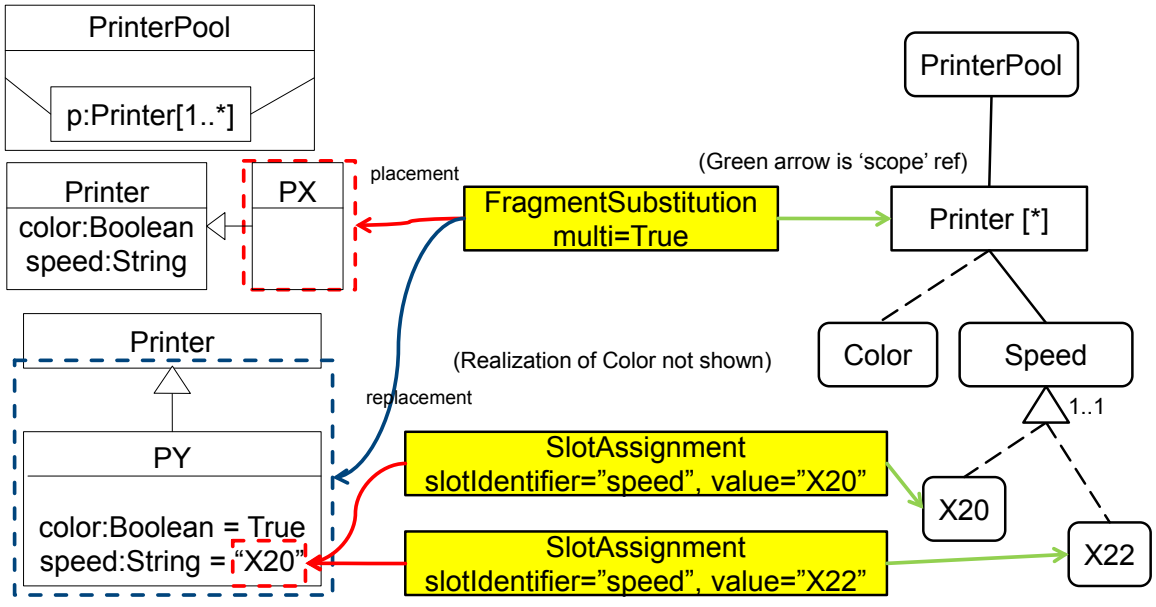



**Figure 32 Variability Realization with Variation Points**

Figure 32 shows the realization of the VClassifier *Printer* and the two Choices *X20* and *X22*. The *Color* choice is realized exactly like the other choices and is therefore omitted here to keep the illustration simpler. The PY class is defined in a library model in addition to the original base model. The significance of such library models is just that they are not included in the resulting product if substitutions do not bring them in.

It is quite obvious and intuitive what the *SlotAssignment* defines, namely, that a given primitive value coded in a string is assigned to a given slot with the name given in another string. MOF reflection is used to find and assign the values. Please compare the application of *ParametricSlotAssignment* in our previous *Scanner* example in Section 7.4.2.

The *FragmentSubstitution* is a little more elaborate than ObjectSubstitution while the principle is quite similar. The Fragment Substitution defines a Placement in the base model which will be substituted by a Replacement. In Figure 32 we indicate the Placement and Replacement, respectively, by a red, dashed frame and a blue, dashed frame on the concrete syntax of the base model. In the model the Placement and Replacement are given by boundary points that contain MOF references to base model elements. Any base model can be seen as a directed graph of MOF objects and it is this property that makes it possible to define the fragments generically as indicated in more detail in Figure 33.

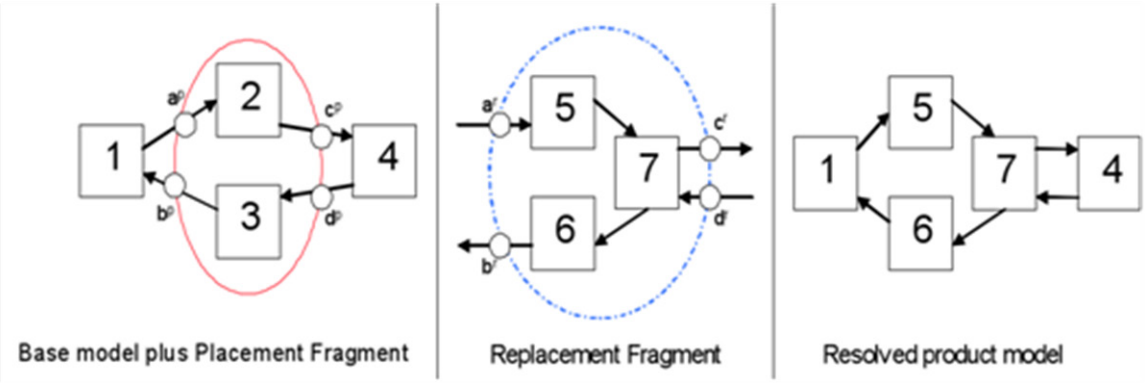



**Figure 33 Details of Fragment Substitution showing Boundary Points and MOF references**

The Fragment Substitution in our example of Figure 32 is defined as being "multi", which means that it can be applied multiple times to the same placement fragment. The first time it is applied, the insides of the placement fragment is removed and the contents of the replacement fragment substituted into that place. The subsequent application of the fragment substitution does not result in any removals, only adding another replacement fragment to the former. This obviously assumes that the multiplicities of the references must allow such addition.

With the resolution model depicted in Figure 31, the fragment substitution is applied twice producing two copies of the PY. The underlying slot assignments will apply to each one of the copies. Notice also that for simplification of the illustrations we have omitted the slot assignment that would give different names to the Printer subclasses. The resulting portion of the product model is given in Figure 34.

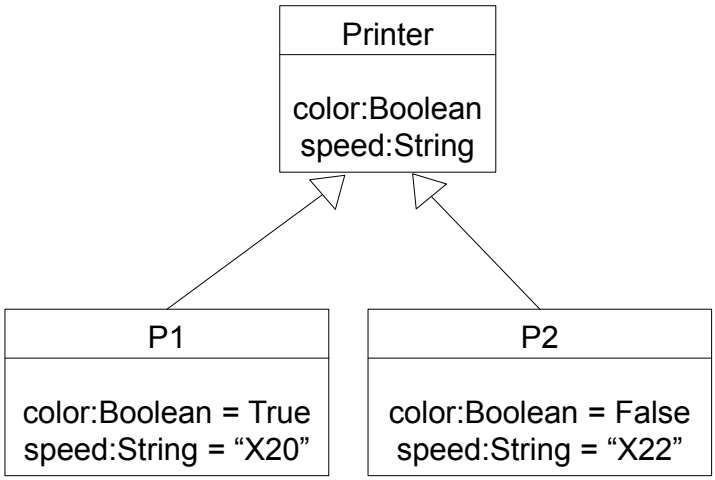



**Figure 34 Portion of the product model of the printer pool with two printers**

The following hierarchical principle holds for the scope of *VClassifiers* and their corresponding repeatable variation points:

- Each VClassifier defines a conceptual scope. In our example the VClassifier PrinterPool defines a conceptual scope.
- Each such VClassifier may have instantiations of replacement fragments associated through the application of a repeatable variation point. Our example has a FragmentSubstitution which is a repeatable variation point and which has two replace fragment instantiations.
- Any VSpec which is a child of the VClassifier corresponds to Variation Points that will change only those elements that are instantiations of replacement fragments of the repeatable variation point of the VClassifier. Our example shows Color and Speed as such VSpecs contained in Printer. Color and Speed are applied to each replacement fragment instantiation, resulting in modifications of P1 and P2 respectively.
- In the imperative interpretation this means keeping a stack of fragment instantiations, and applying a Variation Point can only affect elements on the top of the stack.

### 7.4.4 Opaque Variation Point

In this section, we introduce Opaque Variation Point (OVP) as an executable domain specific (user defined) variation point whose semantics are not defined by CVL; instead a CVL programmer defines the semantic of an OVP using model transformation rules. It should be noted that the semantic specification should contain information about the location of the OVP in the base model along with corresponding base model elements that can participate in the underlying OVP. An example of OVP is described below:

Figure 35 describes a subset of MainPower system where AcmePower and TexasPower are two possible substitution candidates for MainPower system. However, the kind of substitution shown in Figure 35 has a special meaning, i.e., a) candidate power system should inherit all the attributes (in the figure, this is shown as a singleton as defaultAttribute) of the MainPower system and b) a value to the powerType attribute should be assigned based on substituted candidate power system. In particular the classes AcmePower and TexasPower are possible special substitution candidates for class MainPower, where special substitution implies the combination of standard substitution and transformation of the candidate class, i.e. AcmePower or TexasPower. The transformation includes the addition of all attributes of class MainPower to candidate class and default value assignment of attribute “powerType” from name of the candidate class. This requirement cannot directly be achieved through FragmentSubstitution as the semantic of specialized substitution is different than the defined semantic of FragmentSubstitution.

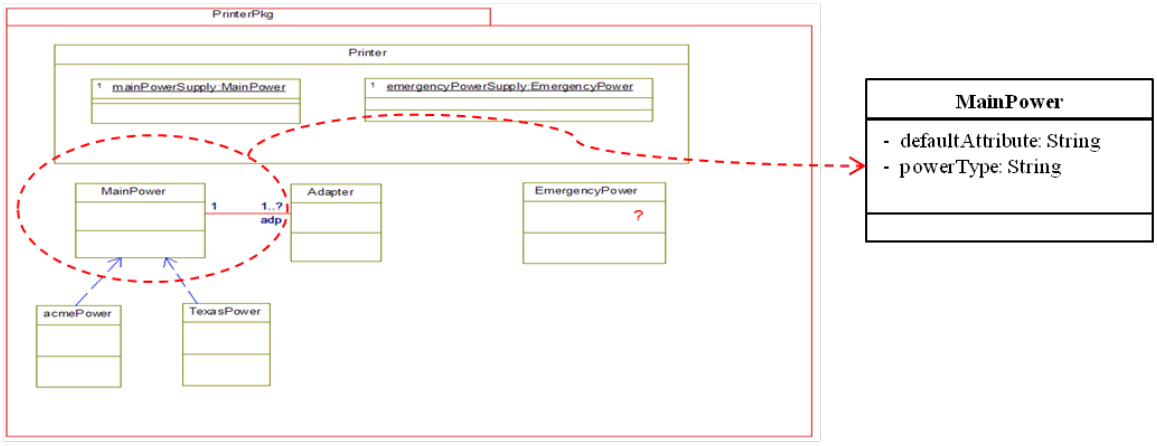



**Figure 35 Attributes of MainPower System**

Figure 36 shows the two possible Opaque Variation Points (OVPs) and their corresponding association with the base model elements. The OVPs are bound to an OVPType, where this type defined the semantic of special substitution explicitly.

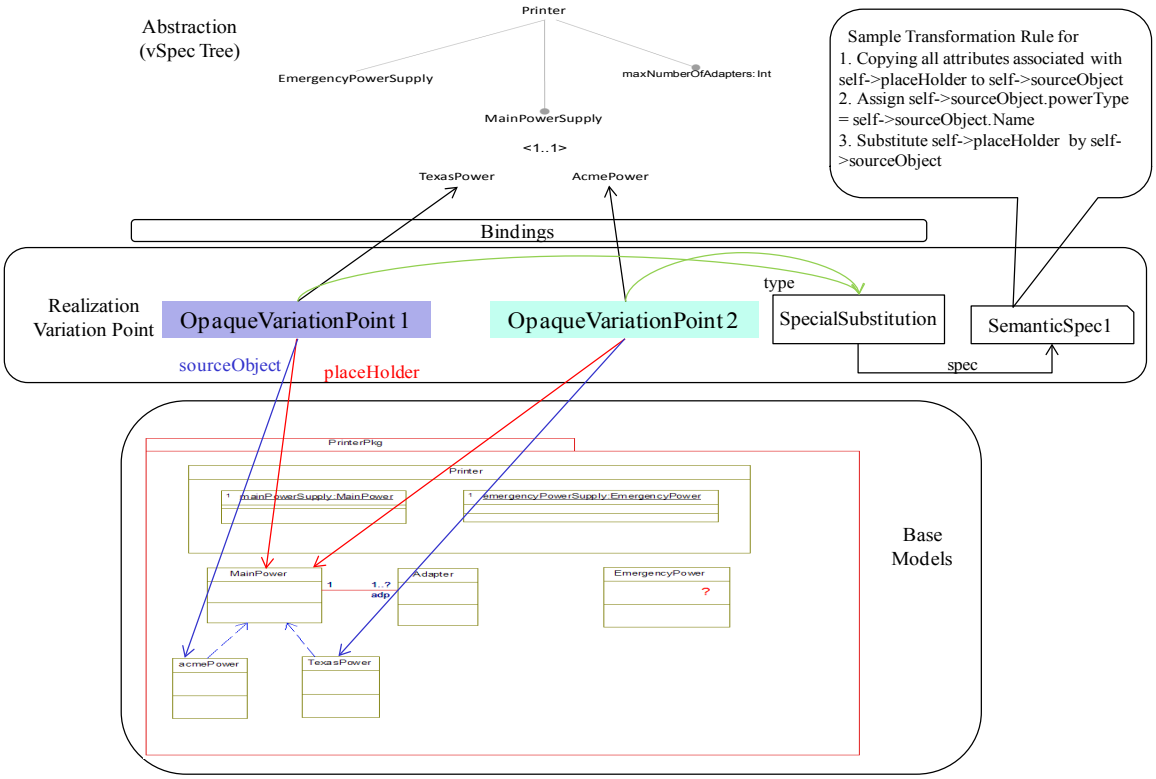



**Figure 36 Variability Requirements for OpaqueVariationPoint**

During variability resolution, the CVL engine should delegate its control to a Model-2-Model (M2M) transformation engine whenever it encounters an Opaque Variation Point. The M2M transformation engine should execute the semantic specification associated with the underlying OVP and resolve variability accordingly. In this context, if OpaqueVariationPoint1 is selected, the underlying M2M transformation engine takes the base model (MainPower and AcmePower) and semantic specification as input and produces the resolved target model as output by executing the user defined specification.

## 7.5 Configurable Units

### 7.5.1 Introduction

The constructs described so far facilitate the specification of variability models comprising logically-organized variation points over base models. These constructs corresponds to the lower layer in the CVL architecture, shown in Figure 6. As shown in the figure, CVL provides on top of that constructs for grouping variability declarations (variation points, VSpecs, and constraints) corresponding to base model containers into *configurable units*.

Using configurable units it is possible to associate a collection of variability declarations with a base model container, such as a UML/SYSML component, package, or class, hide the details of it, and instead expose a *variability interface* through which the entire component can be configured. This facilitates the construction of enterprise-level libraries of reusable components which can be used in several individual projects across the enterprise.

Since CVL is external to the base model, the CVL construct of configurable unit is a standalone element referencing a base model container. Conceptually, however, the pair of base model container and the configurable unit referencing it together define one reusable component, which, in addition to other interfaces it might have (e.g. UML interfaces), also has a variability interface for the purpose of configuration. CVL facilitates the reuse of configurable units both through copying them and then configuring the project-specific copy (the so-called "clone and own" method), as well as through referencing the same library component from several projects, applying a per-usage configuration.

In addition to facilitating reuse, configurable units are also units of *modularization*. For example, a UML/SYSML design for a large enough system will typically contain many packages, components and classes organized in hierarchies, possibly deep ones. By allowing configurable units to be composed of other configurable units, CVL allows to re-iterate the structure of the base model design, thus keeping the modular structure of the project and avoiding the need to maintain a global variability model over an modular base model design.

### 7.5.2 Configurable Units, CVSpecs, and Variability Interfaces

Technically, a configurable unit is a new kind of variation point which references a base model object and by doing so indicates the object is a container with inner variability. The latter is specified via contained variation points. That is, a configurable unit is a kind of variation point which may contain other variation points, i.e. it is a “composite variation point”.

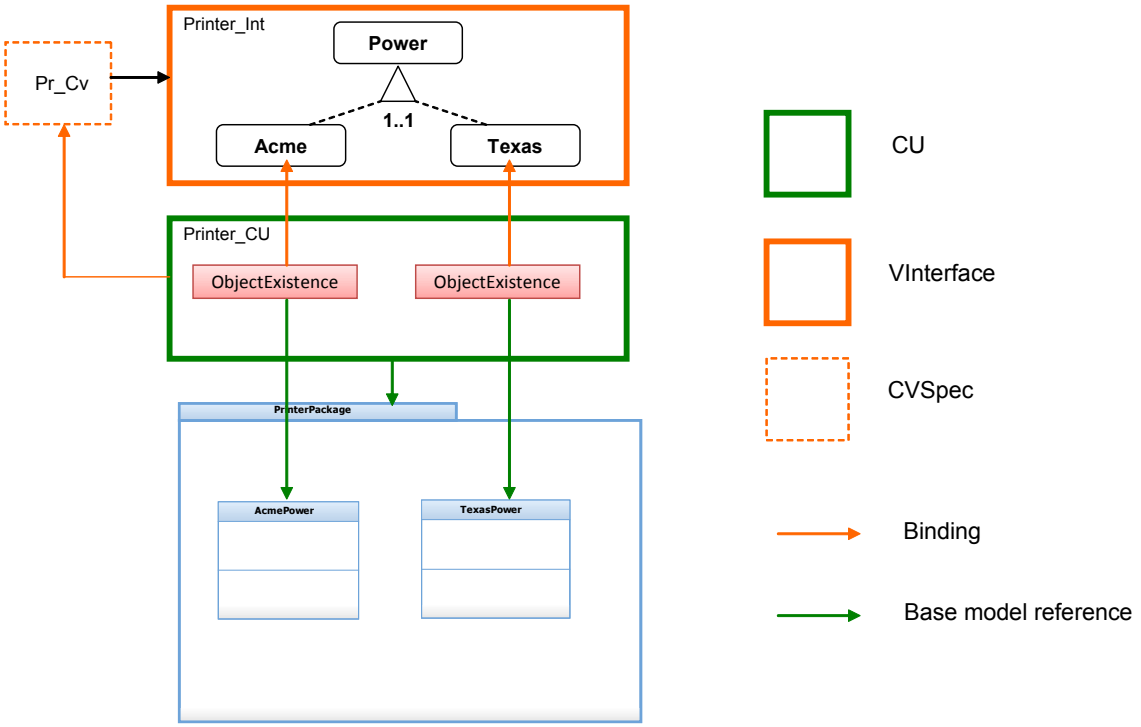



**Figure 37 - A configurable unit bound to a CVSpec**

Figure 37 shows a CVL model defined against a UML base model. The package on the bottom, **PrinterPackage**, contains two classes **AcmePower** and **TexasPower** which model power supply functionality, one made by "Acme" and the other by "Texas". The variability we want to express is that these two classes are alternatives, i.e. in any given product either **AcmePower** or **TexasPower** will be present but not both. Moreover, instead of representing this using "global" variation points we would like to follow the UML structure and relate the variation points to the containing package. In other words we would like to indicate that **PrinterPackage** is *configurable* and specify what it takes to configure it.

This is done by defining a configurable unit **Printer_CU** referencing the package, shown in the center of the figure. **Printer_CU** is a variation point containing other variation points, in this case two Existence variation points against the inner classes, indicating they are optional. Like other variation point, a configurable unit needs to be bound to a VSpec. For this, a new kind of VSpec – *composite VSpec* – is introduced which we abbreviate as *CVSpec*. A CVSpec is a VSpec whose resolution requires resolving a set of VSpecs, identified through its *type*, which is a *variability interface*. A variability interface is just a collection of VSpecs that of course may be organized in tree structures[3].

In Figure 37, the **Printer_CU** is bound to the CVSpec **Pr_Cv** which references the variability interface **Printer_Int**. The Existence variation points inside **Printer_CU** are bound to VSpecs inside that interface, and the tree structure indicates that either the choice **Acme** or the choice **Texas** must be decided positively.

Conceptually, this structure means that **PrinterPackage** is a configurable UML package which exposes a variability interface through which it may be configured. To configure it, clients need not care about the variability internal to the package. All they need to do is provide a resolution for the choices **Acme** and **Texas**, and that will configure the package. The package effectively becomes a reusable component which

[3] By "a collection of VSpecs" we mean informally not just the roots directly owned by the VInterface but also the VSpecs inside the trees, i.e those owned by the roots, those owned in turn by the latter, and so on recursively down to the leaves.

may be configured through its interface. Additionally, the modular structure of the base model is preserved by the CVL variability model.

Mirroring the new kind of VSpec – CVSpec – a new kind of VSpec resolution is also introduced – *variability configuration*, abbreviated as *VConfiguration*. A VConfiguration resolves a CVSpec, and it does so by resolving the VSpecs in its type – a VInterface. Figure 38 shows a VConfiguration **Printer_Texas_Config** resolving the VSpec **Pr_Cv** by deciding **Texas** over **Acme**, resulting in the selection of class **TexasPower** in the base model. The materialization results are shown at the bottom right.

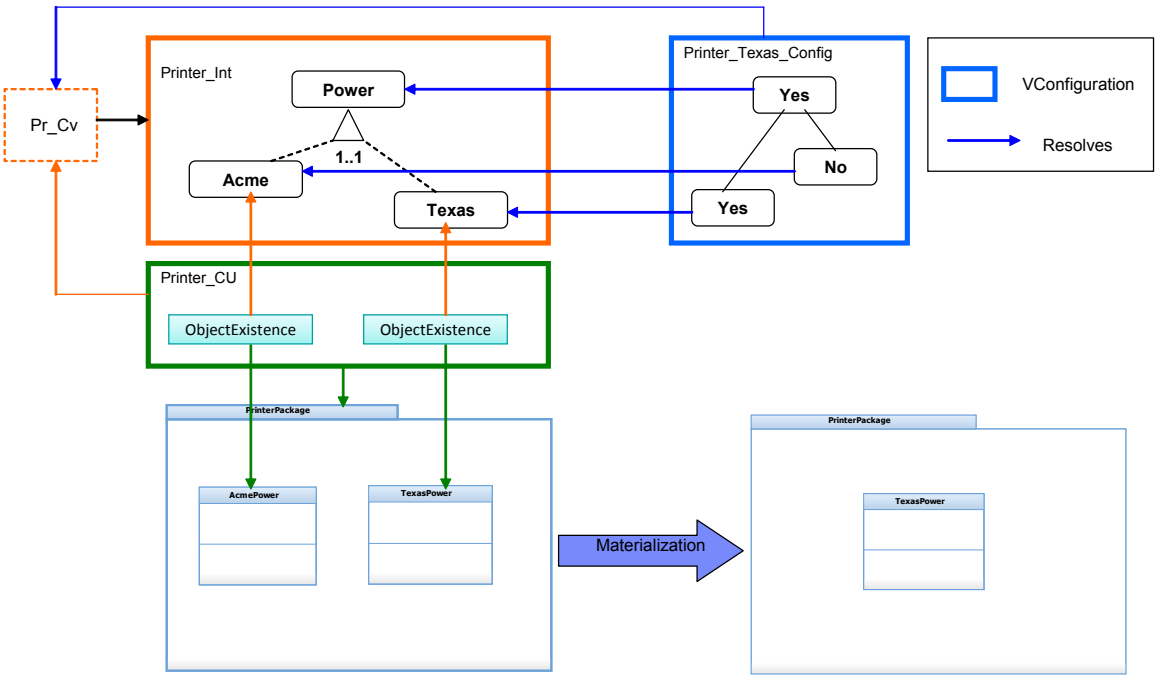



**Figure 38 - VConfiguration resolving a CVSpec**

## 7.5.3 Composition of Configurable units

CVL configurable units may contain other configurable units and so reflect the structure of complex hierarchical base models. Figure 39 builds on the previous example and adds to the base model an enclosing package **Office_Package**, which in addition to the **Printer_Package** from before also contains a new **Scanner_Package**. Each of these two contained packages has inner variability, which is not shown in the figure to avoid cluttering. What we want to express is that in addition to the inner variability in each package, there is variability in the level of **Office_package** which is to select only one of the inner packages in any given product. That is, each office either has a printer or a scanner but not both, plus additionally inner variability in the selected component.

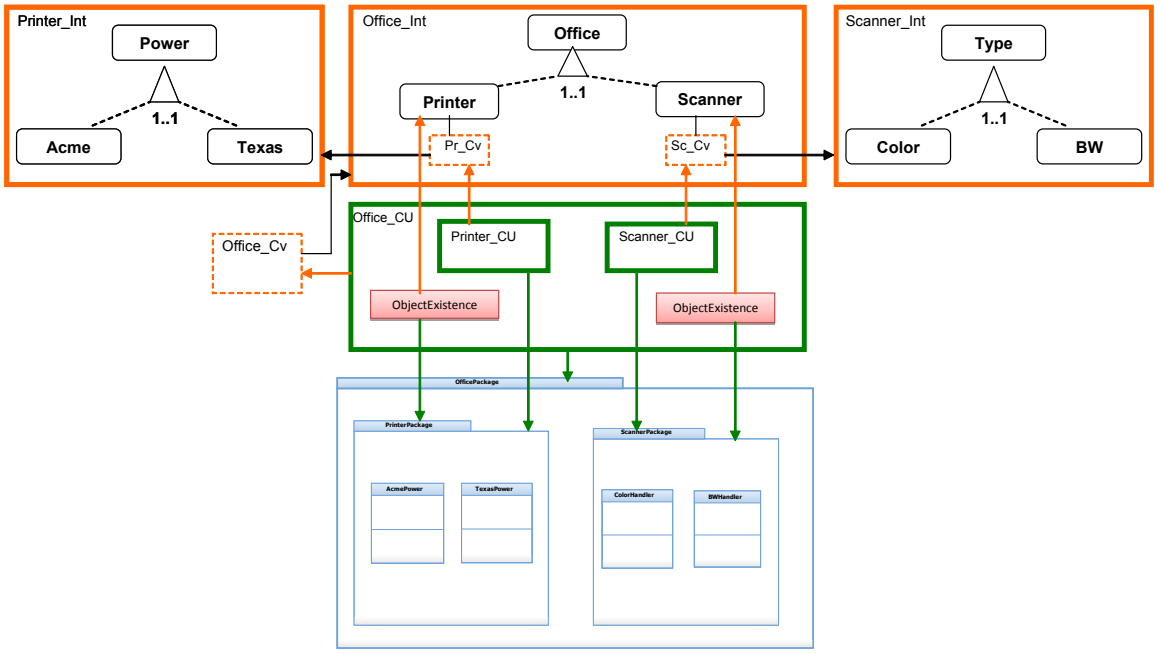



**Figure 39 - Configurable unit composition**

The above is expressed by introducing an enclosing configurable unit **Office_CU** which contains **Printer_CU** and **Scanner_CU**, where each configurable unit references the appropriate base model element. In addition to the configurable units, which are variation points, there are Existence variation points referencing the inner packages. The composite **Office_CU** is bound to a CVSpec referencing an interface **Office_Int** whose VSpec tree involves choices (**Office**, **Printer**, **Scanner**) and CVSpecs (**Pr_Cv**, **Sc_Cv**) to which the inner configurable units are bound.

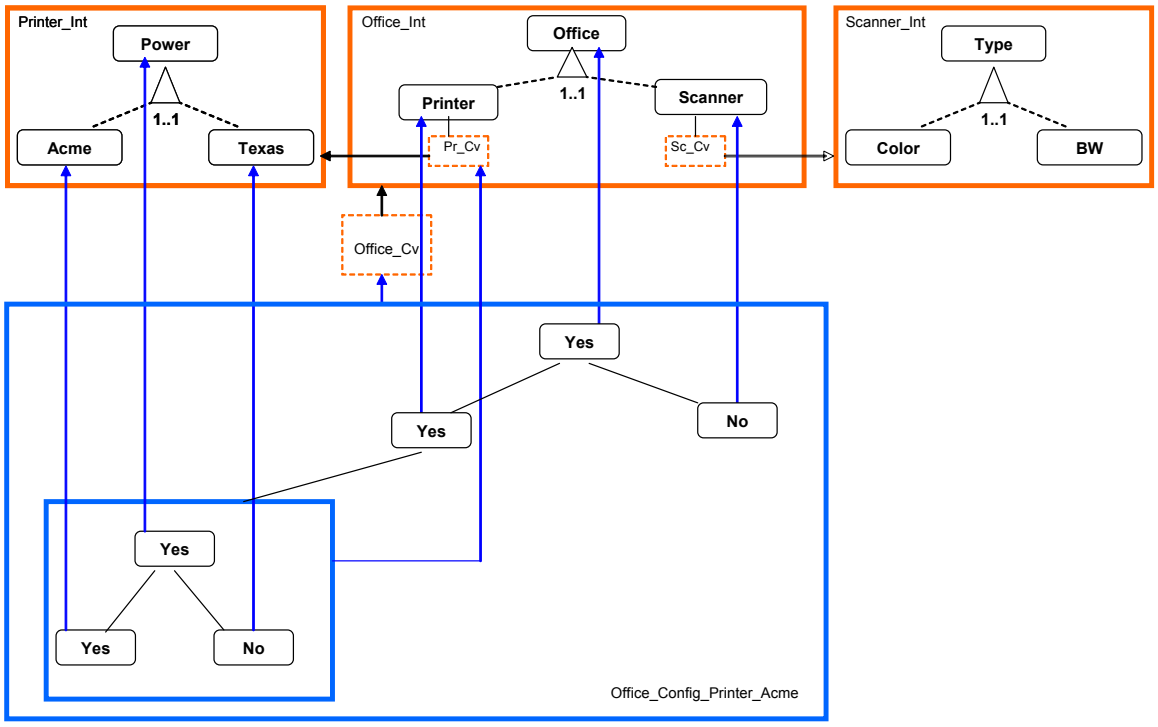



**Figure 40 - A composite VConfiguration**

To materialize **Office_CU** one needs to provide a VConfiguration that resolves the CVSpec **Office_Cv**.

Figure 40 shows such a composite Vconfiguration named **Office_Config_Printer_Acme** which selects **Printer** over **Scanner** and then goes on to select **Acme** over **Texas**.

### 7.5.4 Sharing variability interfaces

Rather than itself being a collection of VSpecs, a composite VSpec is typed by such a collection - a variability interface. The reason for this indirection is to keep variability interfaces as declarative, sharable elements, much like in UML or Java. That is, we would like to facilitate the sharing of a single variability interface among several configurable units.

Figure 41 shows a composite configurable unit **Office_CU** containing two configurable units **InkjetCU** and **LaserCU**, both sharing the variability interface **BW/C_Int**. That is, each is bound to its own CVSpec – **InkjetCU** to **Inkjet_Cv** and **LaserCU** to **Laser_Cv** – but both CVSpecs are typed by same interface, which is about selecting between the choices **BW** and **Color**. Note also how the names of the CVSpecs indicate the *roles* this interface plays in the resolution process: it is used one time to configure an inkjet printer and another time to configure a laser printer.

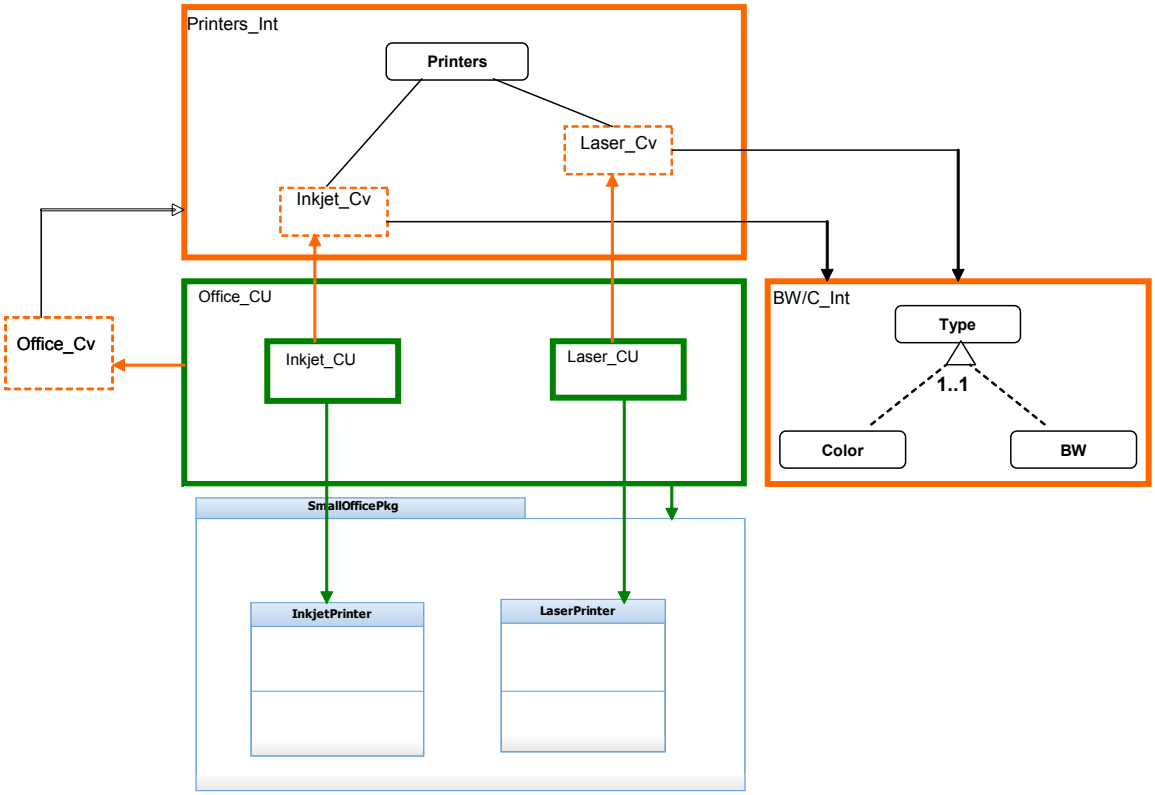



**Figure 41 - Sharing a variability interface**

A composite VConfiguration resolving **Office_Cv** is shown in Figure 42. It contains two sub-configurations, one resolving **Inkjet_Cv** and the other resolving **Laser_Cv**. Those two sub-configurations in fact contain resolutions for the same VSpecs – the choice inside **BW/C_Int** – but resolve them differently – the inkjet printer will be color whereas the laser will be BW.

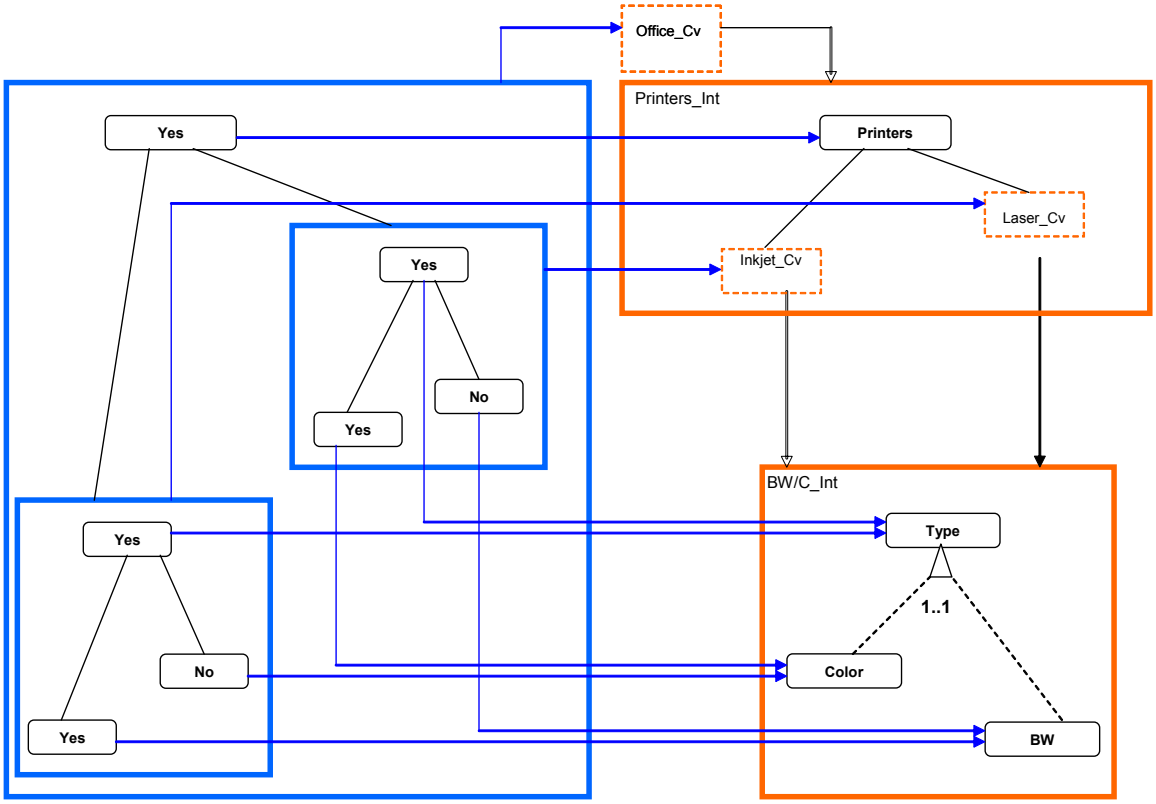



**Figure 42 - A VConfiguration resolving VSpecs differently in different contexts**

### 7.5.5 Composite VSpec Derivation

In the examples so far the CVSpecs of internal configurable units were integrated in the VSpec tree comprising the interface of the enclosing unit. For example, in Figure 41 the CVSpecs **Inkjet_Cv** and **Laser_Cv** are part of the VSpec tree comprising the interface for **Office_CU** (through the CVSpec **Office_Cv**). This results in a conceptual "composite" VSpec tree whose resolution requires a composite VConfiguration like the one in Figure 42. It creates a strong coupling between the interface of the outer unit and that of the inner ones, the latter being conceptually part of the former. This might be the right design choice for some cases, but in other cases we might want to decrease the coupling between the inner units and the outer unit and not integrate the CVSpecs to which the inner units are bound into the interface of the outer unit. The connection between the outer and inner units in such cases would be to *derive* resolutions for the latter from resolutions for the former.

To accommodate this scenario we the concept of *VSpec derivation* discussed in Sec. 7.2.4 is extended to include CVSpec derivations as well. Figure 43 shows a design similar to the previous example, only here the CVSpecs **Inkjet_Cv** and **Laser_Cv** are derived from **Office_Cv**. This is expressed via a *CVSpec derivation*, which is a kind of VSpec derivation. In the figure there are two CVSpec derivations, one deriving **Inkjet_Cv** from **Office_Cv** and another deriving **Laser_Cv** from **Office_Cv**. This is shown as dashed orange arrows. Each such derivation consists of derivations for the members of the referenced interface -- choice derivations in this example. The member derivations are shown as orange labels on the arrows representing the CVSpec derivations. So for **Inkjet_Cv**, **Color** is derived as the value of the **Type1** and **BW** as the value of the **Type2**. For **Laser_Cv** it is vice versa.

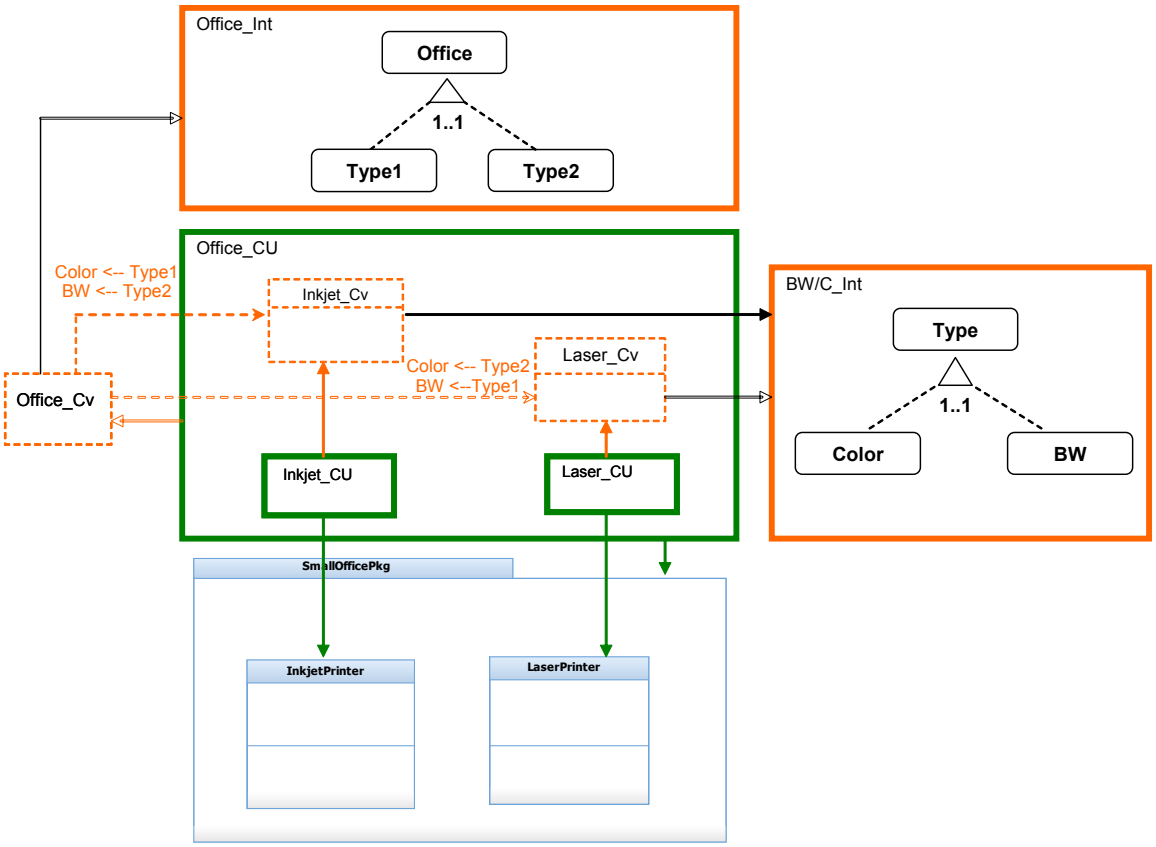



**Figure 43 - Derivation for Composite VSpecs**

### 7.5.6 Using Configurable Units by Reference

Configurable units as discussed so far support component reuse through the so-called "clone and own" method – the component can be copied and then configured as needed. A stronger and more desirable form of reuse is when individual projects can reference components in an enterprise-level library of reusable components, where in each such usage the component is configured for the individual project at hand.

CVL supports this key use-case through a concept of *configurable unit usage*, which technically is just another kind of variation point. In the bottom of Figure 44, the UML Class **Office** has two parts, **mainPrinter** and **backupPrinter**, both typed by class **Printer**. The latter has variability in it which is expressed via the CVL configurable unit **Printer_CU**. In the materialized model, we would like the types of **mainPrinter** and **backupPrinter** to be configured separately, though in the product line model they both reference (i.e. are typed by) **Printer**.

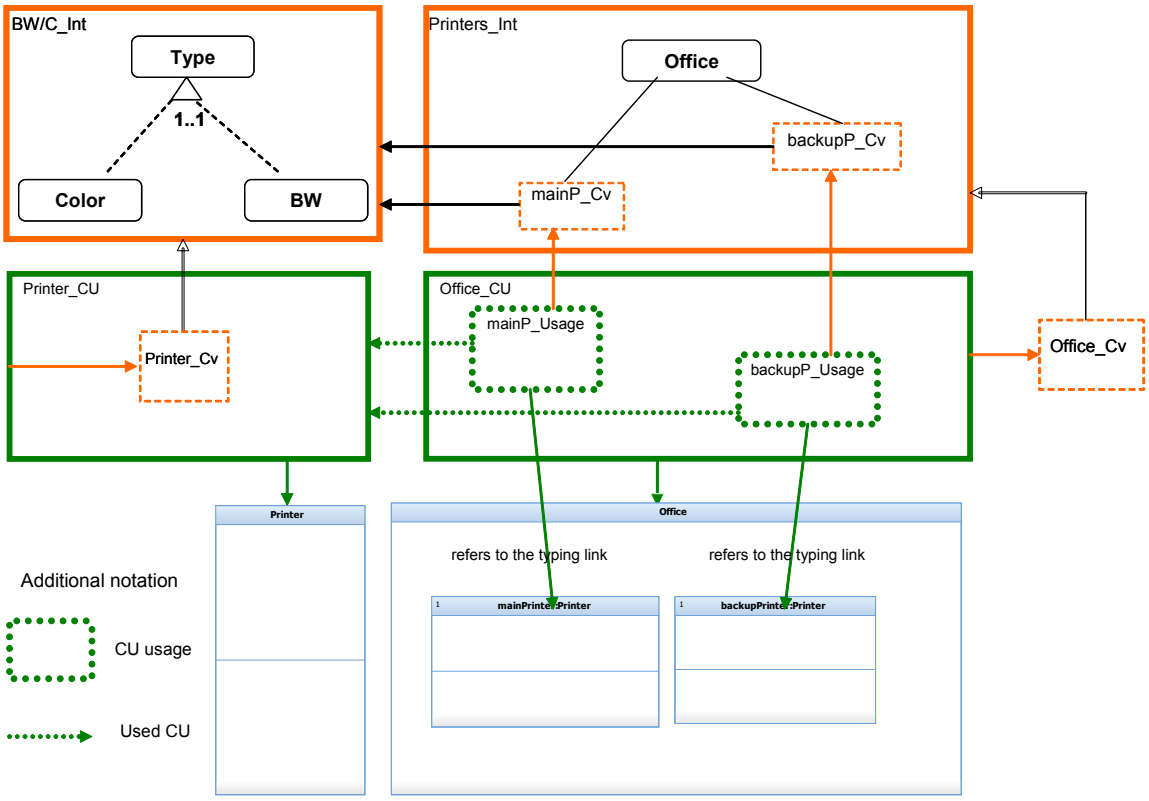



**Figure 44 - Configurable Unit Usage**

To that end we introduce two configurable unit *usages* – **mainP_Usage** and **backup_Usage** -- into **Office_CU**. Both usages reference the **Printer_CU** configurable unit (green dashed arrow) and each is bound to a separate CVSpec. In addition, like all variation points, the usages reference the base model. A configurable unit usage references a *link-end* in the base model representing the usage. The semantics of resolving this variation point would be to deeply clone the unit, i.e. copy all contained elements, elements, they contain, and so on recursively, configure it appropriately, and then redirect the base model link-end to the configured clone. Note that the cloning here is in the semantics, not something the user needs to do.

A configuration for **Office_Cv** is shown in Figure 45. The materialized base model, shown at the bottom, includes two clones of **Printer** – **Printer_Color** and **Printer_BW** – which are then substituted as the types of **mainPrinte**r and **backupPrinter** respectively.

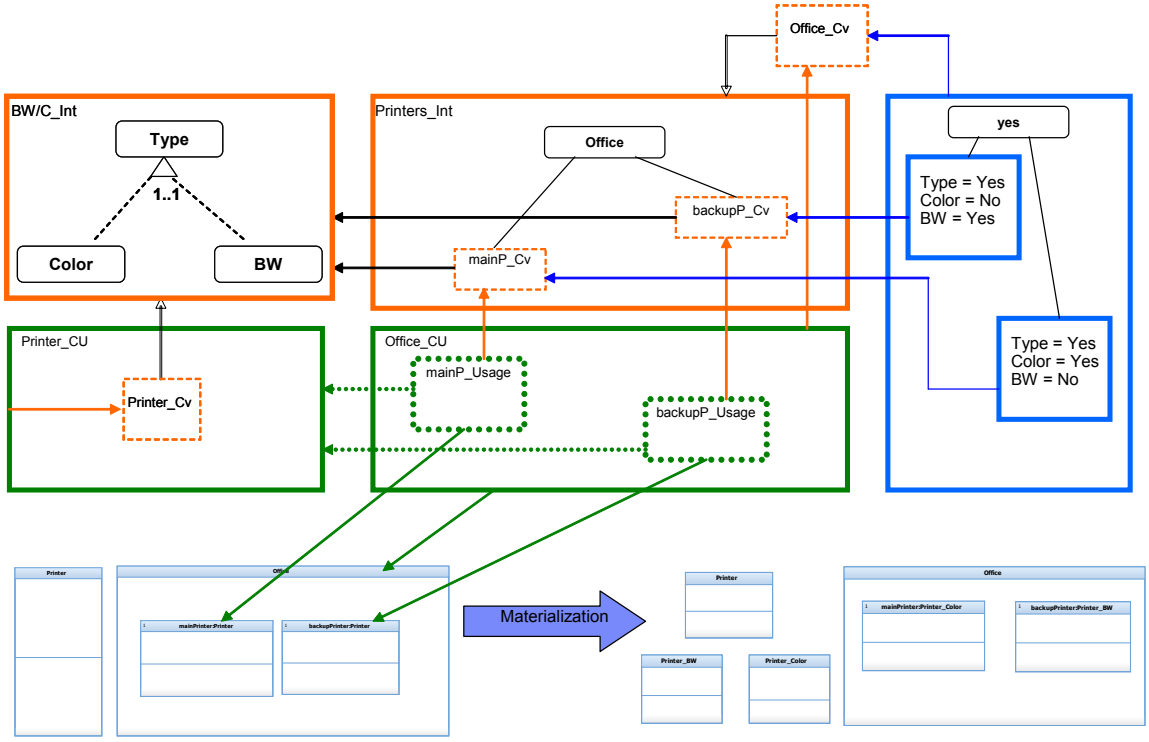



**Figure 45 – Resolving Configurable Unit Usages**

Figure 46 shows an enlarged picture of the materialization, where the different types for **mainPrinter** and **backupPrinter** in the materialized model can be seen clearly.

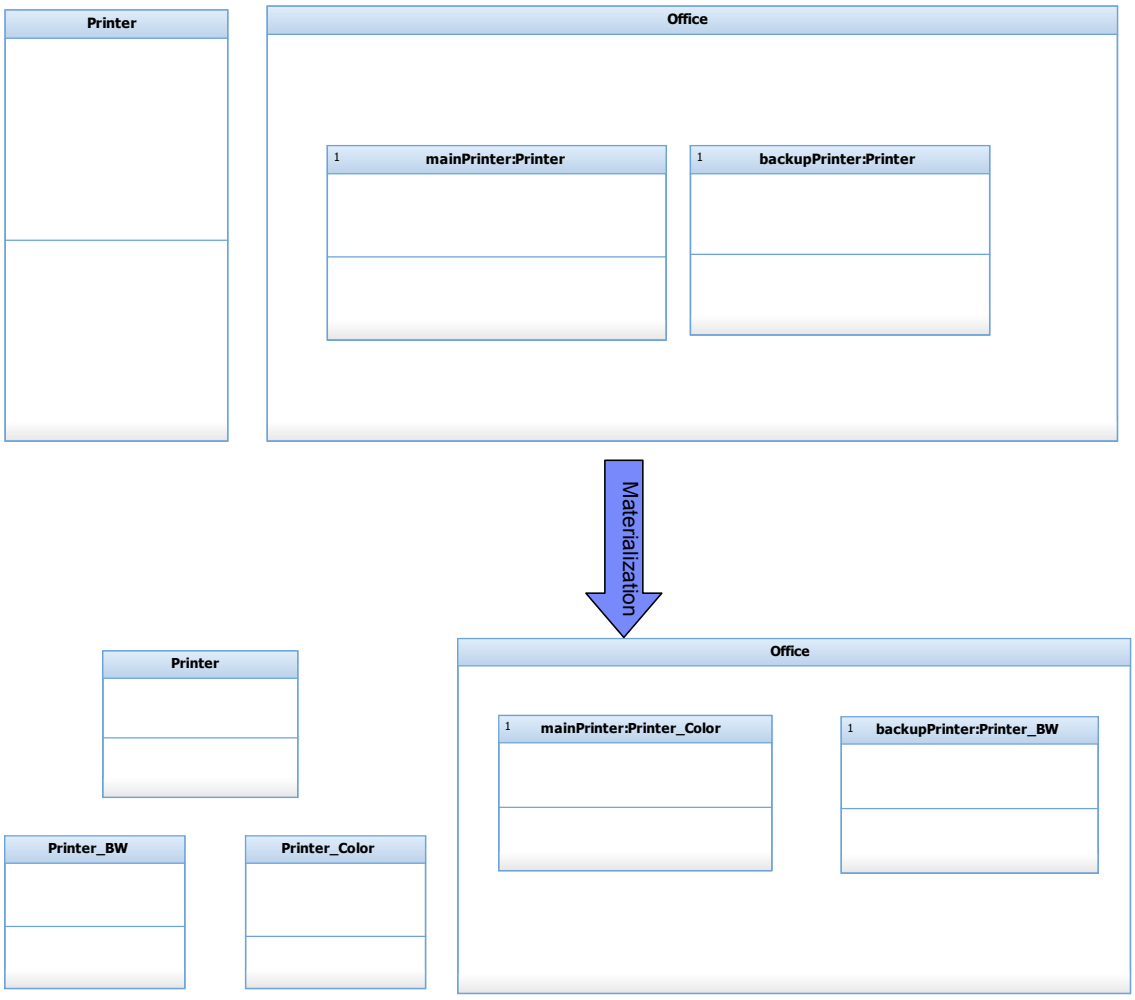



**Figure 46 - Per-usage materialization of a configurable unit**

# 8 Additional Technical Details of the CVL

## 8.1 Overview of the additional technical details

This chapter provides additional details of CVL. The first subsection gives details on Variability Abstraction, defining well-formedness of a resolution model with respect to a VSpec model and the conditions on a resolution model to satisfy its VSpec model. Further, it also defines the syntax and semantics of the basic constraint language. The second subsection deals with Variability Realization, describing the metamodel and the execution semantics of the opaque variation points. Finally, the third subsection deals with Variability Encapsulation, describing how CVSpecs and VConfigurations impact well-formedness and satisfaction of a resolution model.

## 8.2 Abstraction Model

### 8.2.1 Well-Formed and Permissible Resolution Models

This section defines more precisely concepts related to well-formed and satisfying resolution models.

***Def. (VSpec model)*** A *VSpec model* is a collection of VSpecs, i.e. choices, variables, and VClassifiers, a tree relationship between them (there can be more than one root, i.e., in general, the model is a forest), and a set of constraints, each of which is defined in the *context* of some VSpec. Choices have a Boolean attribute **isImpliedByParent** and variables have a **type**.

***Def. (resolution model)*** Given a VSpec model V, a *resolution model* R for V is a collection of VSpec resolutions, i.e., choice resolutions, variable value assignments, and instances, where each resolution in R resolves exactly one VSpec in V (formally, V comes with a resolution mapping R→V). Each choice resolution resolves a choice and has a Boolean **decision** attribute. Each variable value assignment resolves a variable and has a **value** of the variable's type. Each instance resolves a classifier. A resolution model also comes with a tree relationship.

***Def. (positive/negative resolution)*** A Vspec resolution is *positive* if it is either a variable resolution, a classifier resolution (i.e. an instance), or a choice resolution with **decision**=True. Otherwise it is *negative*.

The following definition of a well-formed resolution model guarantees that the tree structure of the resolution model corresponds correctly to that of the VSpec model.

***Def. (well-formed resolution model)*** A resolution model R for V is *well-formed* if the following conditions hold:

1. *The VSpec resolution tree structure mirrors the VSpec tree structure*:
    a. (root) if resolution r in R resolves VSpec v in V and v is a root then r is a root.
    b. (non-root) if resolution r1 in R resolves VSpec v1 in V and v1 is a child of v2 then there is a resolution r2 in R which resolves v2 and r1 is a child of r2.
2. *Uniqueness of choice and variable resolution - different resolutions of the same choice or variable must have different parents*: if r1≠r2 in R both resolve v in V, where v is either a choice or a variable, then v has a parent u and there are p1≠p2 in R which are the parents of r1 and r2 resp., and both p1 and p2 resolve u.

The next concept of a *full* resolution model captures the idea that all needed resolutions are present.

***Def. (full resolution model)*** A well-formed resolution model R for V is *full* if the following hold:

1. All roots are resolved:
    a. Every choice c in V which is a root has a resolution r in R
    b. Every variable v in V which is a root has a resolution r in R
    c. Every VClassifier C in V which is a root has at least n resolutions (instances) in V, where n is the lower instance multiplicity of C
2. All necessary child-resolutions are present: Let r in R be a positive resolution of v in V, then
    a. If c is a child choice of v then there is a resolution r1 in R of c which is a child of r
    b. If u is a child variable of v then there is a resolution r1 in R of u which is a child of r
    c. If C is a child classifier of v then there are at least n resolutions (instances) of C in R which are children of r, where n is the lower instance multiplicity of C
    d. If v has group multiplicity n..m then r has at least n positive child resolutions.

***Def. (Context of a constraint)*** A constraint can have *local context*, which is a VSpec (we refer to it as the *context VSpec*), or it can have a *global context* (i.e., the constraint is global within its enclosing variability package or configurable unit, and it has no context VSpec). The context of a constraint plays two roles: (1) in concrete syntax, it is the starting point for name resolution (cf. Section 8.3.5); (2) in abstract syntax, it determines the VSpec(s) for which the constraint must hold (as specified in condition 4 of the Satisfaction definition below).

A full resolution model as defined above need not be "correct". The following definition of satisfaction captures satisfaction of constraints defined explicitly using the CVL constraint language (Section 8.3.6 defines the satisfaction of such constraints, completing this definition of satisfying resolution models):

***Def. (Satisfaction)*** A full resolution model R for V *satisfies* V if the following conditions hold:

1. Roots are resolved appropriately:
    a. If a root choice c in V has isImpliedByParent=True then there is r in R which resolves c positively
    b. Every VClassifier C in V which is a root has at most m resolutions (instances) in V, where m is the upper instance multiplicity of C
2. *Child resolutions are appropriate*: Let r in R be a positive resolution of v in V, then
    a. If c is a child choice of v with isImpliedByParent = True then there is a positive resolution r1 in R of c which is a child of r
    b. If C is a child classifier of v then there are at most m resolutions (instances) of C in R which are children of r, where n is the upper instance multiplicity of C
    c. If v has group multiplicity n..m then r has at most m positive child resolutions.
3. If resolution r in R resolves choice c in V negatively then all child resolutions of r, if any, are choice resolutions and they are negative
4. *All relevant constraints are satisfied*: All global constraints are satisfied for R. If VSpec resolution r in R resolves VSpec v in V positively then all constraints defined in the context of v are satisfied in the context of r.

***Def. (extension)*** Let R and R' be resolution models for V. R is an *extension* of R' (also R' can be *extended* to R) if R can be obtained from R' by adding VSpec resolutions (along with their attributes and tree relationships).

The following defines what is means for a resolution, even partial, to be correct:

***Def. (permissible resolution model)*** A well-formed resolution model R for V is *permissible* if it can be extended to a full resolution model R' which satisfies V.

## 8.3 Definition of Basic Constraint Language

This section gives an explicit definition of an OCL subset for specifying propositional constraints globally or in the context of VSpecs, including VClassifiers. The constraints relate Choices and Variables using propositional logic operators and operators over Variable types. The section gives the subset's concrete and abstract syntax and semantics. The concrete syntax of the OCL subset is strictly a subset of the concrete syntax of OCL. This subset will suffice for many applications of CVL.

The semantics of OCL constraints outside the subset (Full OCL), such as those involving quantification over instances of VClassifiers, is given by mapping a VSpec hierarchy to a class model and each OCL constraint of the VSpec to an OCL constraint over the class model. Annex 16.2 defines this mapping. The meaning of constraints expressed in the basic constraint language does not change when interpreted via the mapping.

### 8.3.1 Concrete syntax

This section defines the concrete syntax of the OCL subset for basic-level constraints using a context-free grammar in Extended Backus-Naur notation (EBNF). Each subsection defines a non-terminal by one or more productions, gives the corresponding well-formedness rules, and provides the mapping to the abstract syntax element (as defined in Section 8.3.2). The concrete syntax non-terminals end in the suffix "CS" (for concrete syntax).

The following keywords are reserved terminals in CVL and cannot be used as names

‘and’, ‘not’, ‘or’, ‘xor’, ‘implies’, ‘self’, ‘parent’.

#### 8.3.1.1 BCLConstraintCS

An OCL constraint consists of an OCL expression with an optional context specification.
BCLConstraintCS ::= ('context' VSpecCS 'inv :')? BclExpressionCS

**Mapping to abstract syntax:** BCLConstraintCS maps to BclConstraint. An OCL expression, which maps to the association end bclExpression in the abstract syntax of the basic constraint language, is optionally preceded by context declaration (which maps to the association end context in the abstract syntax).

#### 8.3.1.2 BclExpressionCS

An OCL expression is either OperationCallExpCS or VSpecCS or LiteralExpCS; each of these expression types is defined in subsequent sections.

BclExpressionCS ::= OperationCallExpCS
BclExpressionCS ::= VSpecCS
BclExpressionCS ::= LiteralExpCS

**Mapping to abstract syntax:** BclExpressionCS maps to BclExpression.

#### 8.3.1.3 VSpecCS

VSpecs are referred to by name or via name paths.

VSpecCS ::= <String>
VSpecCS ::= VSpecCS '.' <String>

**Well-formedness:** The <String> expression in the first production must be a VSpec name, or special a name, i.e., *self* or *parent*. In case of navigation (the second production), the right hand side name of the dot expression must be a child of the left hand side or the special variable *parent*. VSpec names are composed of letters, digits, and underscores. Name cannot start with a digit.

**Mapping to abstract syntax:** The navigation expression is mapped to a single VSpecRef. Name resolution rules unambiguously determine the referenced VSpec. The Name Resolution Rules, see Section 8.3.5, specify how VSpecCS maps to VSpecRef.

#### 8.3.1.4 LiteralExpCS

LiteralExpCS ::= <Number>
LiteralExpCS ::= "'"<String>"'"
LiteralExpCS ::= BooleanLiteralExpCS

**Well-formedness:** In the first production <Number> must be real number, unlimited natural, or integer number. It is given as a <String>.

**Mapping to abstract syntax:** LiteralExpCS maps to LiteralExp.

#### 8.3.1.5 BooleanLiteralExpCS

BooleanLiteralExpCS ::= 'true'
BooleanLiteralExpCS ::= 'false'

**Mapping to abstract syntax:** BooleanLiteralExpCS maps to BooleanLiteralExp, where BooleanLiteralExp.booleanSymbol represents the literal (*true* or *false*).

#### 8.3.1.6 OperationCallExpCS

OperationCallExpCS ::= OperationUnCS BclExpressionCS
OperationCallExpCS ::= BclExpressionCS OperationCS BclExpressionCS
OperationCallExpCS ::= BclExpressionCS '.' FunName '(' ArgumentsCS? ')'

**Mapping to abstract syntax:** OperationCallExpCS maps to OperationCallExp. Note that the single argument for unary operations, both arguments for binary operations, and the target of a function call also map argument—an association end in the abstract syntax.

#### 8.3.1.7 ArgumentsCS

ArgumentsCS ::= BclExpressionCS ( ',' ArgumentsCS )?

**Mapping to abstract syntax:** ArgumentsCS maps to argument—the association end in the abstract syntax.

#### 8.3.1.8 OperationUnCS

OperationUnCS ::= 'not' | '-'

**Mapping to abstract syntax:** The terminal 'not' maps to the corresponding element in the Operation enumeration.

#### 8.3.1.9 OperationCS

OperationCS ::= 'and' | 'or' | 'implies' | 'xor' | '+' | '-' | '*' | '/' | '=' | '<=' | '>=' | '<' | '>'

**Mapping to abstract syntax:** Each terminal in the production maps to the corresponding element in the Operation enumeration.

#### 8.3.1.10 FunName

simpleName ::= 'conc' | 'isDefined' | 'isUndefined'

**Mapping to abstract syntax:** Each terminal in the production maps to the corresponding element in the Operation enumeration.

### 8.3.2 Abstract syntax

The following diagram shows an excerpt from the CVL abstract syntax showing the part representing the basic constraint language.

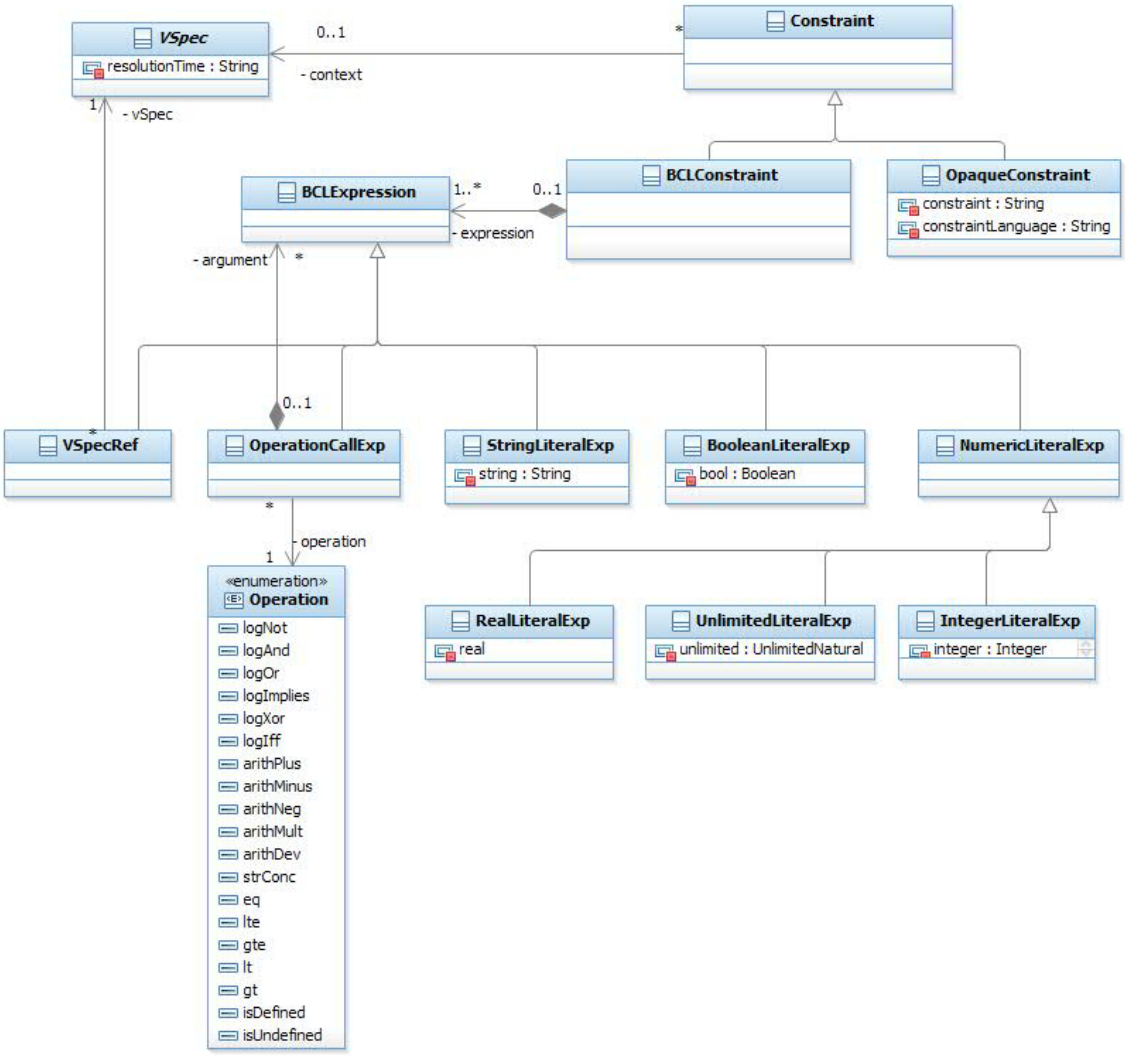



**Figure 47 Metamodel of the constraint language**

Below we provide several examples of the constraints presented in Section 7.3, visualized as object diagrams conforming to the abstract syntax meta-model in Chapter 11.

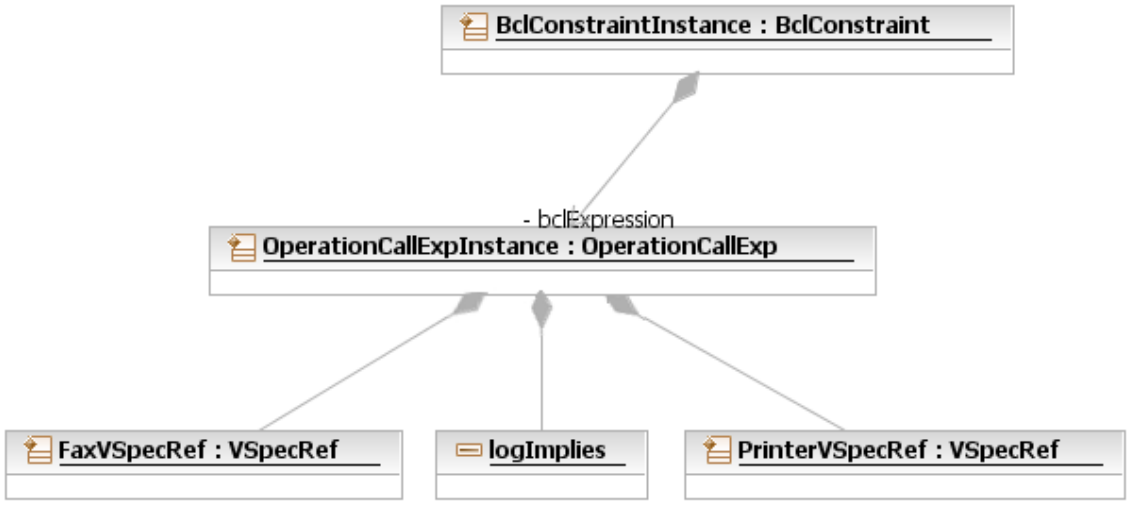



**Figure 48 Abstract syntax of the simple propositional constraint from Figure 9**

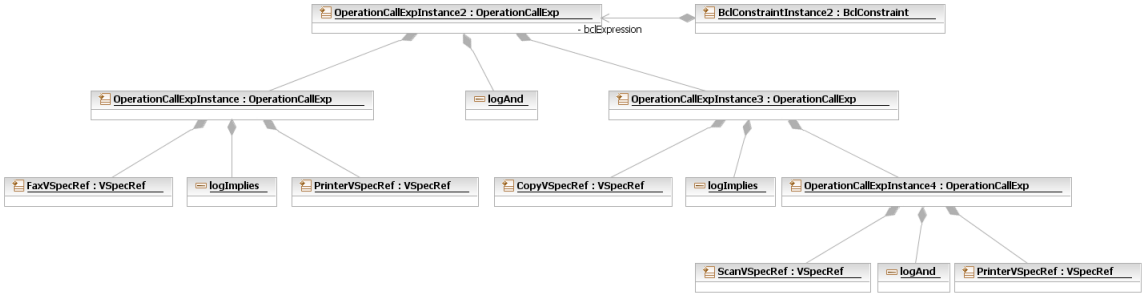



**Figure 49 Abstract syntax of the complex propositional constraint from Figure 10**

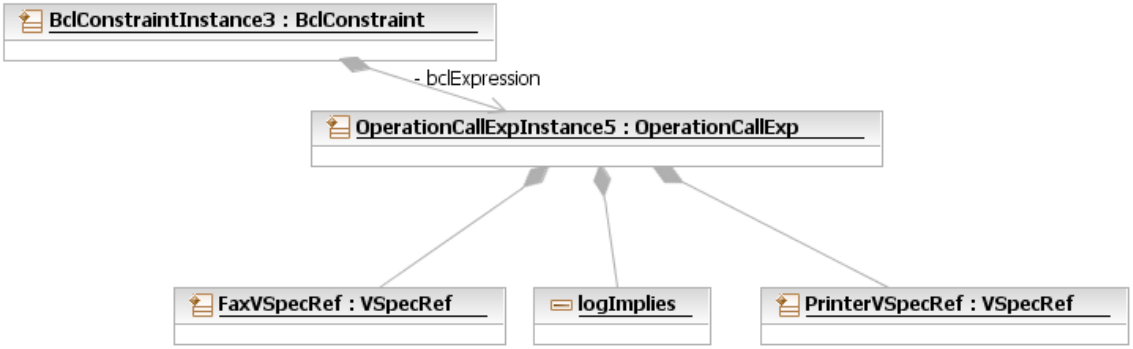



**Figure 50 Abstract syntax of the constraint with a path expression from Figure 12**

A path expression, such as printerPool.fax, always resolves to a unique VSpec. Therefore, in the abstract syntax, the path expression reduces to a VSpecRef element (see Figure 50).

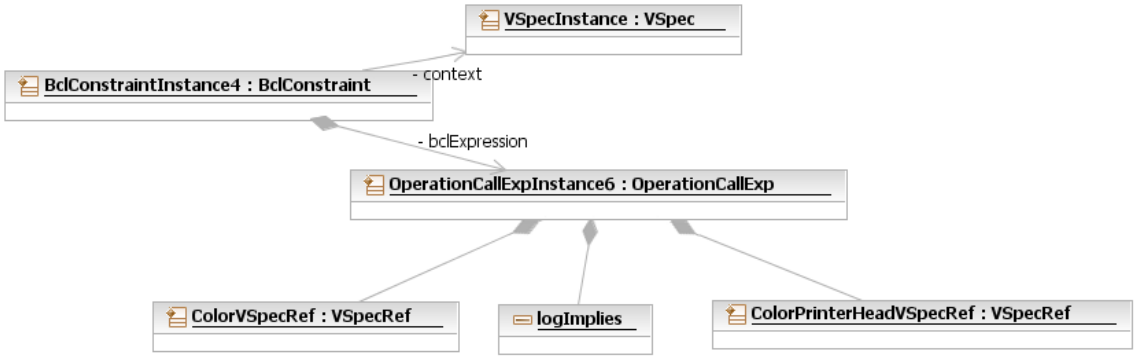



**Figure 51 Abstract syntax of the constraint with implicit quantification from Figure 16**

In the implicit quantification example in Figure 16, the constraint is specified under the Printer classifier. The constraint must hold for each instance of the classifier. The concept of implicit quantification is not manifested explicitly in the meta model: it is given implicitly via the context (all instances of the context classifier have to satisfy the constraint). The context of the constraint in the example from Figure 16 is determined based on the graphical notation (see Figure 51).

### 8.3.3 Typing Rules

Typing rules are specified over abstract syntax. We assume that there is a global environment with all the VSpecs from VSpec tree. Environment is a map from expressions and VSpecs to types. The remainder of this section defines the typing rules. We use the following conventions: $\tau$ represents a type (see PrimitiveTypeEnum in the meta-model), c represents an BclConstraint, exp represents an BclExpression and v represents a VSpec.

*(1) OCL Expressions Evaluate to Boolean Values*
Each OCL constraint contains an OCL expression that evaluates to a Boolean value.

$$\frac{c : \text{BclConstraint}}{c.\text{bclExpression} : \text{Boolean}}$$

*(2) Unary Logical Operators*
The only unary logical operator is logical negation. Negation of a Boolean expression is of Boolean type.

$$\frac{exp : \text{Boolean}}{\text{logNot}(exp) : \text{Boolean}}$$

*(3) Function Application*
Function of type $\tau 1 \rightarrow \tau 2$ applied to expression of type $\tau 1$ evaluates to expression of type $\tau 2$.

$$\frac{f : (\tau 1 \rightarrow \tau 2),\ exp : \tau 1}{f(exp) : \tau 2}$$

*(4) Binary Logical Operators*
A binary logical operator takes two Boolean expressions and evaluates to a Boolean value.

$$\frac{exp1 : \text{Boolean},\ exp2 : \text{Boolean}}{}$$

log(exp1,exp2) : Boolean

*(5) Unary Arithmetic Operators*
The only unary arithmetic operator is arithmetic minus.

$$\frac{\text{exp : Number}}{\text{arithMinus(exp) : Number}}$$

*(6) Binary Arithmetic Operators*
An arithmetic operator takes two expressions that evaluate to numbers, and evaluates to a number.

$$\frac{\text{exp1 : Number, exp2: Number}}{\text{arith(exp1,exp2) : Number}}$$

*(7) String Operation*
The only string operation is string concatenation. String concatenation takes two String expressions and evaluates to String.

$$\frac{\text{exp1 : String, exp2: String}}{\text{strConc(exp1,exp2) : String}}$$

*(8) Relational Operators*
A relational operator takes two expressions of the same type and evaluates to a Boolean value.

$$\frac{\text{exp1 : } \tau\text{, exp2: } \tau}{\text{rel(exp1,exp2) : Boolean}}$$

*(9) isDefined Operation*
The isDefined operation takes expression of any type and evaluates to a Boolean value.

$$\frac{\text{exp : } \tau}{\text{isDefined(exp) : Boolean}}$$

*(10) isUndefined Operation*
The isUndefined operation takes expression of any type and evaluates to a Boolean value.

$$\frac{\text{exp : } \tau}{\text{isUndefined(exp) : Boolean}}$$

*(11) Choice*
A choice yields Boolean type.

$$\frac{\text{v : Choice}}{\text{v : Boolean}}$$

(*12) Variables*
A VSpec variable has type τ that defines the type of valid values (e.g. Integers, Strings, etc.). Thus, a variable yields its type τ.

$$\frac{v : \text{Variable}}{\tau}$$

### 8.3.4 Well-Formedness Rules

A constraint can refer only to choices and variables, but not to classifiers. Further, the closest ancestor classifier of a VSpec referenced in a constraint, if such classifier exists, must be either the context—if the context is a classifier, or must be the same classifier as the closest ancestor classifier of the context. If the constraint is global, then the referenced VSpecs must not have any ancestor classifier.

### 8.3.5 Name Resolution Rules

Name resolution rules disambiguate names of VSpecs used in constraints. A name is resolved in the context of a VSpec (context is the end of the association between BclConstraint and VSpec that touches VSpec in the metamodel) or globally in up to five steps.

1. Check if it is a special name *self* or *parent*. Semantically, *self* indicates a resolution of the VSpec in whose context the constraint is specified. Thus, this name resolves to the context VSpec. The special name *parent* maps to the *parent* of the resolution that resolves the VSpec that is the context of the constraint; thus, this name is resolved to the parent of the context VSpec. If the context is a root VSpec, so it has no parent, the constraint is not well-formed (an error). Similarly, if the context is global, neither *self* nor *parent* are allowed.
2. Look up the name in children VSpecs of the context VSpec in breadth-first search manner. If the constraint is global, start the lookup in the root VSpecs of the VSpec model.
3. Search in the ancestors VSpecs starting from the parent VSpec of the context and up. For each ancestor, look up the name also in its descendants.
4. Search in other top-level VSpecs. For each VSpec apply rule (2).
5. If the name cannot be resolved or is ambiguous within a single step (for rules 3-4), the constraint is not well-formed and an error is reported.

For navigations (expressions of the form $v_1.v_2...v_n$) the name resolution rules are applied to resolve $v_1$. Once it is resolved, subsequent VSpecs ($v_2...v_n$) are resolved by applying only rules (1), (2), and (5). Note that $v_1$ becomes the context VSpec for resolving $v_2$, and $v_2$ becomes the context for $v_3$, etc.

### 8.3.6 Semantics

This section defines the semantics of constraints. In particular, it defines how to determine whether a constraint C is satisfied for a given *full resolution model* R and the optional context resolution r (the constraint is global if r is missing). The semantics works with the abstract syntax of the constraint, where navigation paths and special names (self, parent) have already been resolved to specific VSpecs. At this stage, the context resolution r, if present, is used to determine the classifier instance, if any, under which r is nested. This is necessary since a VSpec that is nested under a classifier may have multiple resolutions and r is used to select the right one.

The semantics uses two interpretation functions, each named *I* but with different signatures. Both interpret abstract syntax elements, but one additionally takes a context resolution model as a parameter. The first function interprets types and operations. The other takes the context resolution model and an expression. Note that a constraint is an expression, just a Boolean-valued one.

The result of evaluating an expression can be undefined. The resolution hierarchy mirrors the corresponding VSpec hierarchy, but allows for missing resolution subtrees. In particular, child resolutions of negative choice resolutions can be missing. Asking for a missing resolution yields the value *Bottom*, which a special value representing "undefined." Thus, the evaluation of an expression can yield Bottom.

The *satisfaction function S* checks the satisfaction of a constraint (thus, the Definition of Satisfaction in Section 8.2.1 can call this function to check the satisfaction of a constraint). The satisfaction function takes a given resolution model R, the optional context resolution r (or the special value Bottom, if the context resolution is missing), and the constraint C (in abstract syntax), and returns True, False, or Bottom. The constraint C is satisfied if and only if S(R,r,C) is True.

The satisfaction function S is defined in terms of the interpretation function over expressions (this interpretation function is defined in the following subsections) as follows:

S(R,r,C) ≜ I(R',C), where R' is (1) the subtree of R rooted in r if r is a classifier instance, or (2) the subtree of R rooted in a classifier instance that is the closest parent of r if r is defined (i.e., r ≠ Bottom) and such instance exists, or (3) R otherwise.

Note that, by the well-formedness rules defined in Section 8.3.4, R' will contain all the resolutions needed to evaluate C and every VSpec that C refers to will have at most one resolution in R'.

#### 8.3.6.1 Basic types

The semantics of a basic type is given by the interpretation function I mapping each type (represented by an abstract syntax element) to a set:

- I(Integer) ≜ Z ∪ {Bottom}
- I(Real) ≜ R ∪ {Bottom}
- I(Boolean) ≜ {True,False} ∪ {Bottom}
- I(String) ≜ A* ∪ {Bottom}

Z represents the set of integers; R represents the set of real numbers; and A* represents the set of all finite strings over the alphabet A. Note that each set is extended with the Bottom value to account for the possibility that the value of an expression is undefined.

#### 8.3.6.2 Operations

The CVL logical operators map to the standard three-value logic operators, with their semantics defined by Table 1.

I(logNot) ≜ ~
I(logAnd) ≜ &&
I(logOr) ≜ ||
I(logImplies) ≜ =>
I(logXor) ≜ xor

**Table 1 Semantics of Boolean operations in three-value logic**

| $b_1$ | $b_2$ | $\sim b_1$ | $b_1$ && $b_2$ | $b_1$ \|\| $b_2$ | $b_1$ => $b_2$ | $b_1$ xor $b_2$ |
|---|---|---|---|---|---|---|
| False | False | True | False | False | True | False |
| False | True | True | False | True | True | True |

| | | | | | | |
|---|---|---|---|---|---|---|
| True | False | False | False | True | False | True |
| True | True | False | True | True | True | False |
| False | Bottom | True | False | Bottom | True | Bottom |
| True | Bottom | False | Bottom | True | Bottom | Bottom |
| Bottom | False | Bottom | False | Bottom | Bottom | Bottom |
| Bottom | True | Bottom | Bottom | True | True | Bottom |
| Bottom | Bottom | Bottom | Bottom | Bottom | Bottom | Bottom |

The CVL arithmetic operators (including inequality comparison operators) map to their corresponding mathematical counterparts over Z and R extended with the case when any of their arguments evaluate to Bottom, in which case the result is Bottom, too. Similarly, srtConc maps to the concatenation function over two strings, also extended with the case when any of their arguments evaluate to Bottom, in which case the result is Bottom

The equal operator (eq) maps to, as expected, value equality comparison; if any of the arguments evaluates to Bottom, the result of comparison is Bottom, too. Note that this operator is also used to express logical equivalence.

Applying isUndefined to Bottom yields True; applying it to any other value yields False. Conversely, applying isDefined to Bottom yields False; applying it to any other value yields True.

#### 8.3.6.3 Expressions

The semantics of expressions is given by the interpretation function I that takes a context resolution model and the abstract syntax representation of a CVL expression. The resolution model plays the role of an environment where the resolutions of the VSpecs referred to by the constraint can be looked up. The function returns a value that is element of I(Integer) ∪ I(Real) ∪ I(Boolean) ∪ I(String).

The following subsections define this function for the case of operation expressions (application of operators), VSpec references, and literals.

#### 8.3.6.4 Operation Expressions

The interpretation of applying a CVL operation to a set of argument expressions is defined by applying the interpretation of the operation to the interpretations of the argument expressions:

I(R, Operation(a1, a2, …, an)) ≙ I(Operation)(I(R, a1), I(R, a2), …, I(R, an))

#### 8.3.6.5 VSpec references

The basic constraint language allows references to two types of VSpecs: choices and variables. Note that we assume that R belongs to a fully resolved resolution model.

For a choice reference c:
I(R, c) ≙ True if the resolution of c* in R is positive (c* is the choice that c refers to)
I(R, c) ≙ False, otherwise (i.e., if the resolution of c* in R is negative or if it is missing).

For a variable reference v:

I(R, v) ≜ val if there is a resolution of v* in R and its value is val (v* is the variable that v refers to).
I(R, v) ≜ Bottom otherwise (i.e., a resolution of v* in R is missing)

#### 8.3.6.6 Literals

For the CVL literal true of type BooleanLiteralExp, I(_,true) ≜ True; similarly, I(_,false) ≜ False.

We assume that analogous mappings between CVL literals that are instances of StringLiteralExp, RealLiteralExp, IntegerLiteralExp, and UnlimitedNaturalExp and their corresponding semantic values are also defined.

## 8.4 Variability Realization Model

### 8.4.1 Opaque Variation Point

This section describes in detail the metamodel and the execution semantics of Opaque Variation Point (OVP). An OVP is a ChoiceVariationPoint whose semantic is not defined by CVL but it is expected that the semantic will be defined by the CVL Programmer while defining Variability Realization Model.

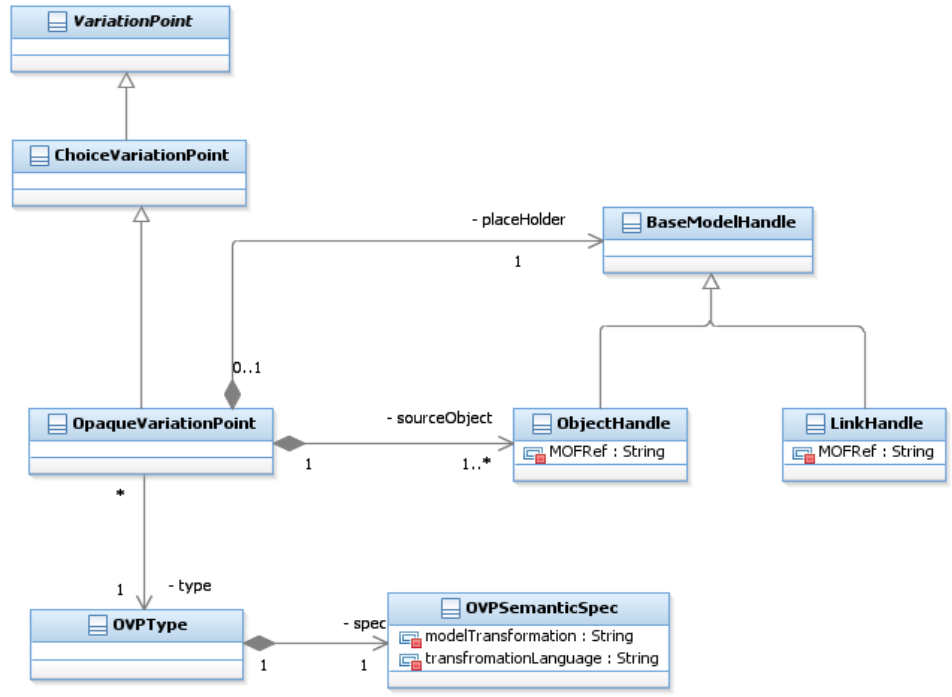



**Figure 52 Opaque Variation Point Metamodel**

As a specification, an OVP refers to a placeHolder object and a set of sourceObjects. The placeHolder determines the location in the base model where model transformation rule needs to be applied and a set of sourceObjects decides the set of model elements in the base model that can participate in the transformation rule. An ObjectHandle or LinkHandle can be a candidate for the placeHolder. As depicted in the metamodel, each OPV is characterized by an OVPType (opaque variation point type) for reusability

purpose (see Opaque Variation Point 7.4.4 for a specific example). Each OVPType defines a specific variation point semantic using OVPSemanticSpec (captured as a string in the metamodel). The OVPSemanticSpec should conform to any M2M transformation language (e.g. QVT and Kermeta).

#### 8.4.1.1 Execution Semantics of Opaque Variation Point

The CVL Engine should execute any Opaque Variation Point in accordance to the following execution semantics:

- CVL engine should delegate the control to a M2M Transformation engine if OpaqueVariationPoint needs to be executed
- Specific M2M transformation engine is selected based on the transformationLanguage attribute of OpaqueVariationPoint→OVType→OVPSemanticSpec.
- M2M transformation engine fetches the semantic specification from the modelTransformation attribute of OpaqueVariationPoint→OVType→OVPSemanticSpec.
- The transformation rules are applied on the associated base model element referred by OpaqueVariationPoint→placeHolder.
- The transformation rules uses a set of base model elements referred by {OpaqueVariationPoint→sourceObject} as its input.

## 8.5 Variability Encapsulation

The definitions in Section 8.2, including well-formed and satisfying resolution models, and the definition of the basic constraint language, are easily extended to handle variability encapsulation concepts. For the purpose of these definitions, a CVSpec can be treated like a choice with impliedByParent being true and its children being a copy of the root VSpecs form the VInterface that the CVSpec points to. The CVSpec also receives a copy of the global constraints from the VInterface, and the CVSpec becomes the context of these constraints.

# 9 CVL Concrete Syntax

The concrete syntax of CVL is not fully defined. This document provides a definition of notation for variability abstraction which covers what has been traditionally known as "feature diagrams". We do realize that even variability abstraction can benefit from other description techniques such as matrices (tables), but for many purposes a graphical notation suited for tool manipulation and with improved clarity compared with the earlier dialects of feature diagrams will be useful. There is also a directly corresponding notation for the resolution model.

For the variability realization constructs we have decided that since realization is tightly coupled to the base language and the corresponding tooling, we will leave this to the tools. Still we provide some examples of how the realization information can be expressed.

No concrete syntax is suggested for the Configurable Units.

## 9.1 Concrete Syntax of Variability Abstraction

The concrete syntax for basic Variability Abstraction is shown by the example in Figure 53.

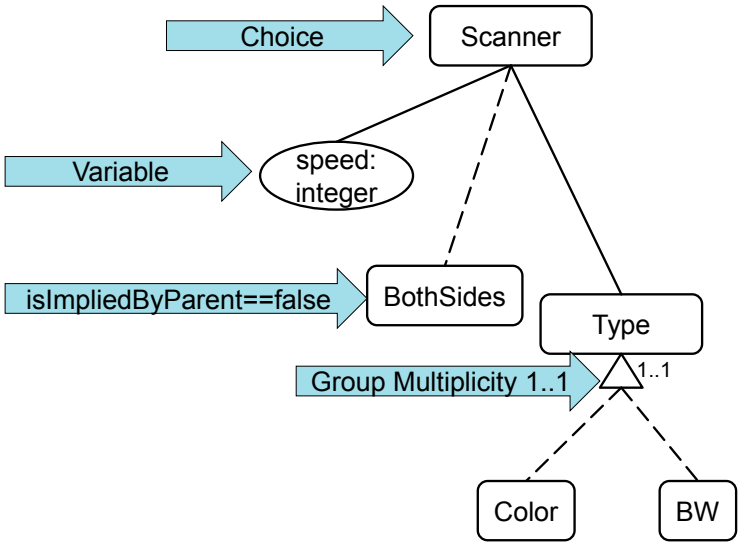



**Figure 53 Notation for basic variability abstraction**

We have annotated the Vspec diagram with arrows indicating the concepts that are depicted by the given symbols.

A *Choice* has a rounded rectangle as its symbol with its name inside.

A *Variable* is represented by an oval (ellipsis) containing the variable name and its type separated by a colon.

A dashed link between choices indicates that the child choice is not implied by its parent. This corresponds to a flag in the Choice metaclass where *isImpliedByParent* is **false**. When the flag is **true**, the child is resolved according to its parent. Thus, in our example BothSides can be either resolved to true or false, while Type must follow the resolution of Scanner which means if there is a Scanner it must have a Type. The latter corresponds to what is modeled as mandatory features, while the former corresponds to modeling optional features.

In the Scanner example in Figure 53, the Scanner choice has no *Group Multiplicity*.

The Type choice, however, has a GroupMultiplicity with lower bound equaling 1 and upper bound equaling 1. This means that there can only be one and only one child of Type which is resolved positively. Both Color and BW are dashed, but both choices cannot be resolved positively in the same resolution. Neither can there be a resolution with neither Color nor BW. The Group Multiplicity is indicated by a triangle placed just below the VSpec (here *Type*) from which links are connected to the children (here: *Color* and *BW*). The lower and upper bounds are indicated by a text placed close by the triangle where the lower and upper bounds are separated by two points (here: 1..1).

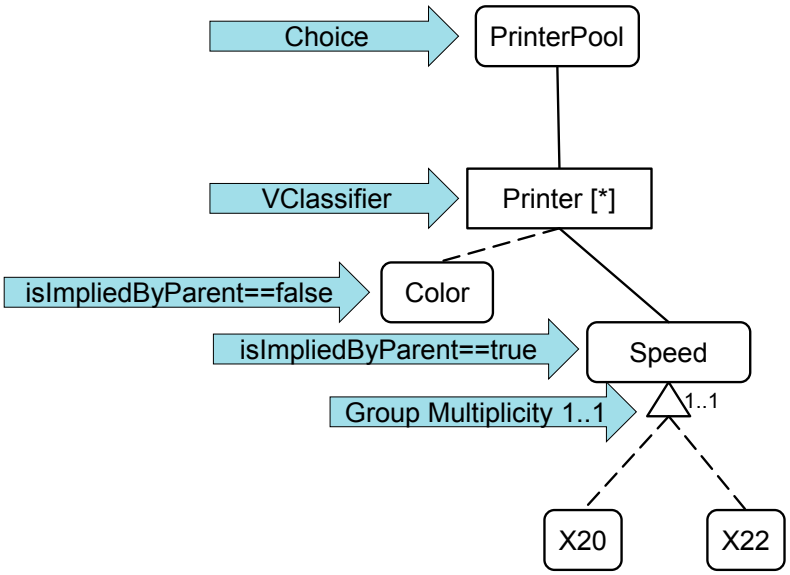



**Figure 54 Notation for VClassifier**

In Figure 54 we show the notation for VClassifiers.

*VClassifier* is described by a rectangle with sharp corners containing the name of the VClassifier followed by the *Instance Multiplicity* in brackets. The instance multiplicity constrains the number of instances of the VClassifier. In our example the Instance Multiplicity is an asterisk which indicates from zero to many instances of Printer.

Notice that VClassifiers are not shown to be optional with a dashed link because the instance multiplicity describes its legal cardinality.

On the other hand VClassifiers can have group multiplicity and have any kind of Vspec children.

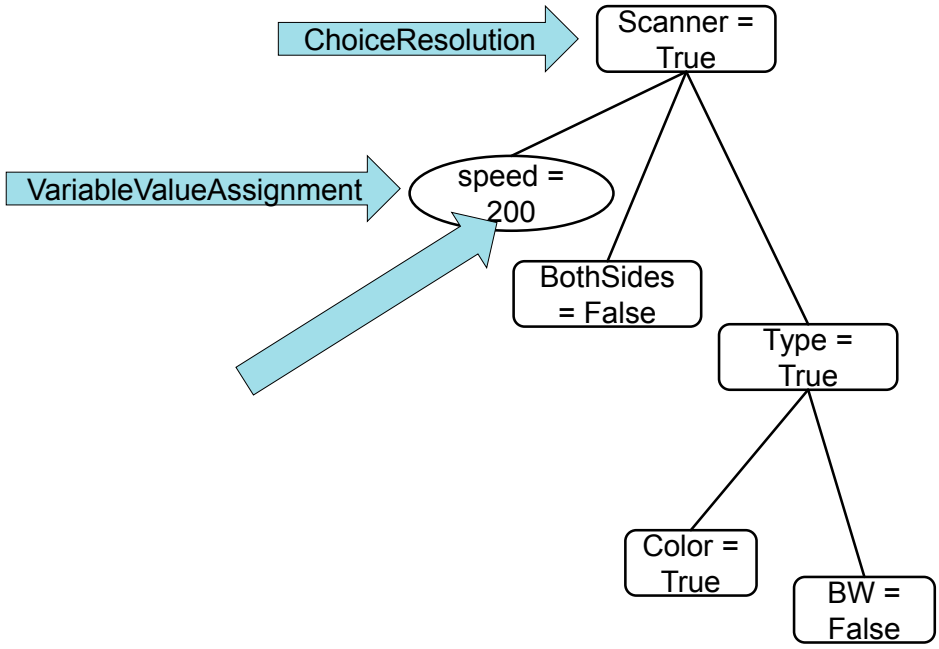



**Figure 55 Notation for resolution model of basic variability abstraction**

The notation for the resolution model is almost identical to that of the VSpec model as shown in Figure 55. We apply the same symbol for *ChoiceResolution* as we did for Choice in the VSpec model, namely a

rounded rectangle. The decision is given after an equal sign.

For variables the resolution is a *VariableValueAssignment* where the *ValueSpecification* is given following an equal sign.

The resolution model is a structural replica of the corresponding VSpec model, but we do not need the opional (dashed) links and the group multiplicities as they represent constraints that need to be satisfied by the resolution model. The names of the VSpecs are repeated in the resolution model diagram for the simplest possible correspondence.

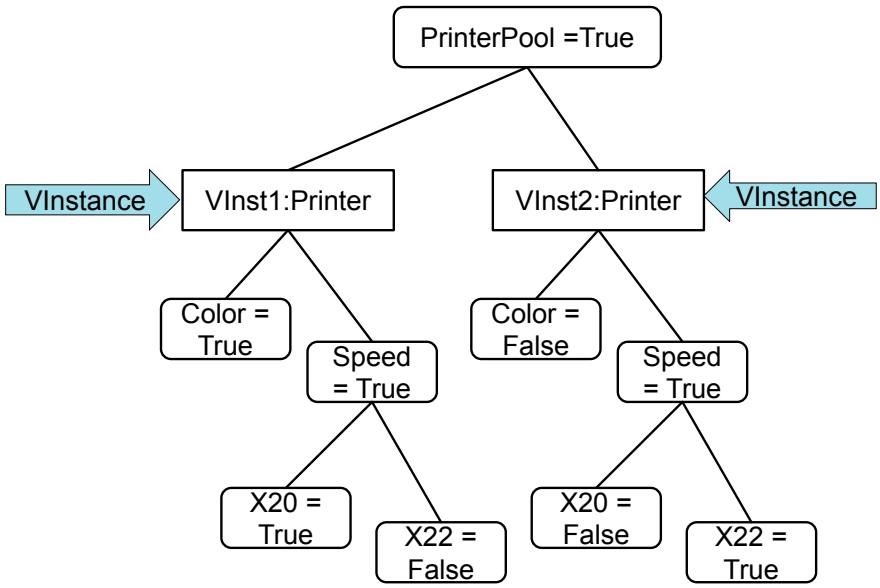



**Figure 56 Resolution model for VClassifiers**

In Figure 56 we see how resolution models for VClassifiers are described. Each *VInstance* can be named and is identified and resolved individually. The VInstance symbol is the same as that of VClassifier, but it contains an instance name followed by a colon and the name of the corresponding VClassifier.

## 9.2 Variability Constraints

Above we have only considered notation relating to the VSpecs themselves. Supplementing the VSpecs are the Constraints which just constrain which resolutions are legal. The Constraints are given in parallelograms that are linked to a VSpec representing the context of the constraint.

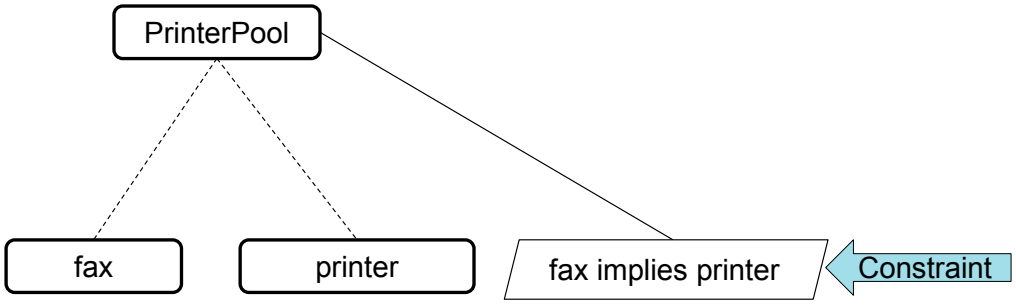



**Figure 57 Simple constraint**

In Figure 57 we show en example of the simplest possible constraint. The *Constraint's* context is the PrinterPool and it simply says that if your resolution selects a fax it also needs a printer. The constraint is necessary and it is cumbersome to express this without the constraint.

The context is significant when a VClassifier is the context as shown in Figure 58.

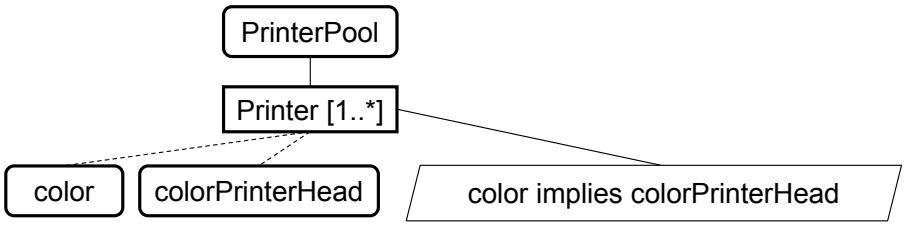



**Figure 58 Constraint in a VClassifier**

In Figure 58 the constraint expresses that for every Printer resolution, if it has color, then it will need a colorPrinterHead.

The constraints we have covered to now is expressed in the constraint language which is integrated into the CVL language and metamodel. If the CVL user needs to go beyond these first order constraints, he/she needs to apply full OCL and relate to the OCL standard and textual notation.

# 10 Semantics of Variability language

This Chapter will be used to explain the principles of the Semantics and explain this through an example. This chapter presents the semantics of CVL, illustrated on our running example. The chapter's focus is on the dynamic semantics of CVL, *i.e.*, what happens to the base model during materialization according to a resolution model. The static semantics, i.e., well-formedness rules of the abstract syntax, is given in Chapter 11.

## 10.1 CVL Dynamic Semantics

### 10.1.1 Specification of dynamic semantics of CVL

We begin with Variability Abstraction and Variability Realization. The semantics of variability encapsulation units will be defined subsequently. The dynamic semantics of each metaclass is detailed further in the **CVL Manual**. An example of dynamic semantics code is also available in the annex **Variation point semantics example in Kermeta (see section 16.1)**.

#### 10.1.1.1 Dynamic semantics of Variability Abstraction and Variability Realization

##### *10.1.1.1.1 Auxiliary definitions for CVL dynamic semantics*

We recall some useful definitions to define CVL dynamic semantics (for more details about these definitions please refer to **7.1 Introduction** and **8.2.1 Well-Formed and Permissible Resolution** :

- **Base model:** the model to which variability information is attached using CVL. It is a graph of model elements conforming to an arbitrary meta-model specified in MOF.

- **Variability model:** the CVL model (i.e.: variations points, variability specification and constraints) used to declare variability on a base model. Since a CVL model, from the viewpoint of dynamic semantics, has no predefined *root* (possible roots include a variability package and a Configurable Unit), we assume that materialization starts with a set of VSpecs and a set of VSpecResolutions (contained in the resolution model) that are accessible in the context of the CVL execution.

- **Resolution model:** this model stores the resolution of the *variability model* through a tree of VSpecResolutions such as ChoiceResolutions determining for each choice whether it is resolved positively or negatively and VariableValueAssignments determining values assigned to variables.

- **Resolved model:** a new model derived from the **base model** by materialization according to a resolution model.

- **Solution space:** the set of all possible **resolved models** with respect to a given **base model** and a given **variability model** (including its constraints). In our case, we can define two kinds of variability: a positive variability and a negative variability. Positive variability means that we add base model elements associated with a positively decided choice into the resolved model, whereas negative variability means we remove base model elements associated with negatively decided choices. Note that if the variability model contains positive variability, the solution space might be infinite.

Note these two important considerations:

- **The aim of deriving a resolved model from a base model and a variability model (for a given resolution model) is to reduce the solution space cardinality.** This derivation is obtained by considering a variability model as a "program" parameterized by the resolution model and operating on the base model, to provide a resolved model. Initially, the **resolved** model is equal to the base model. Then the execution of each "statement" of the variability model adds new constraints on the solution space, hence progressively reducing its cardinality, eventually down to 1 to get a fully resolved model, or to 0 if there are inconsistencies in the CVL model.

- **Containment issues and well-formedness of the resolved model.** The CVL dynamic semantics does not impose anything about containment links or cardinality constraints (and other well-formedness rules) in a resolved model.
  For example, if the execution of a given choice leads to delete a state S1 from a Finite State Machine base model, CVL semantics neither impose nor prevent handling the resulting dangling transitions. CVL semantics is defined in such a way that it only says *"state S1 cannot be present in the resolved model"*. It thus allows any tool to do more, i.e., to follow containment links and/or remove additional parts of the resolved model to fulfill its well formedness rules defined through its meta-model constraints. Any CVL compliant tool is thus free to add more constraints on the solution space than what is defined here for the execution of resolution choices. These constraints should normally follow from the properties of the metamodel for the base model.

#### *10.1.1.1.2 Specification of dynamic semantics by intention*

- **Global view on the CVL dynamic semantics**
  In the following, we specify the CVL dynamic semantics by intention. Since the CVL semantics is defined operationally for each cvl element as adding new execution operations on the solution space, the rest of this section gives the pre and post-condition of the execution of each Variation Point metaclass of a CVL model. These constraints are defined using OCL pre and post conditions on an "eval" operation defined as follows:

  void eval(in ctx : CVLExecutionContext) .

It is assumed that the resolution model is organized as a tree of VSpecResolution that is iterated over in a depth first manner to call the relevant *eval* functions. Thus, for a given VSpecResolution we start with the evaluation of all of its children (and their children) and then we evaluate its sibling. However, before evaluating the resolution model, in order to calculate resolutions that are not explicitly expressed in the resolution model, we evaluate the constraints of the model and create the implicit resolutions.

- **Semantics environment**
  To make the description of CVL dynamic semantics easier to read, we group the required elements of the semantics environment into a CVLExecutionContext classifier.
  Figure 59: CVL Execution Context, gives an overview of this classifier.

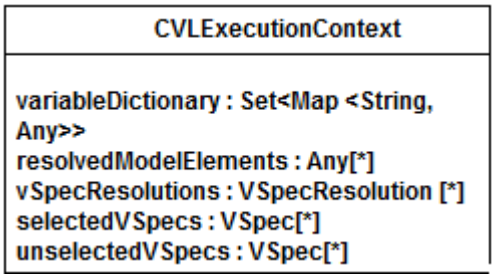



**Figure 59: CVL Execution Context**

- **CVLExecutionContext#variableDictionary**: the dictionary to manage variables (a set of Map<variable name, value> to be also used with VClassifier). It is used to retrieve the values of values.

- **CVLExecutionContext#resolvedModelElements**: the set of model elements in the base model after the materialization.

- **CVLExecutionContext#vspecResolutions**: the set of VSpecResolution from the given resolution model we want to use during the materialization.

- **CVLExecutionContext#selectedVSpecs**: the set of the positively resolved VSpecs, according to the VSpecResolution set.

- **CVLExecutionContext#unselectedVSpecs**: the set of the negatively resolved VSpecs, according to the VSpecResolution set.

In the initial state, most of these collections are empty, except resolvedModelElements. Then, the execution semantics is expressed by populating different sets of the CVLExecutionContext classifier. The **resolvedModelElements** collection is initialized with the set of model elements from the base model.

#### *10.1.1.1.3 Dynamic semantics pre- and post-conditions*

The dynamic semantics of each CVL metaclass are detailed in Chapter **11**, the **CVL Metamodel** in the paragraphs on dynamic semantics. For each metaclass representing a CVL element, we present the pre- and post-condition of its execution, first in an informal way and then using OCL.

An example of operational semantics written in Kermeta can be found in the annex **Variation point semantics example in Kermeta,** in section **16.1**. Kermeta is a language workbench designed for specifying and designing domain-specific languages (DSL). The purpose of using Kermeta is to weave dynamic semantics into the CVL Metamodel, complementing the static semantics provided by OCL.

#### 10.1.1.1.3.1 Variability Abstraction and Variability Resolution

For this part, the dynamic semantics is detailed on the CVL Manual.

#### 10.1.1.1.3.2 Variation point semantics (Variability Realization)

We further present the eleven variation points that inherit respectively from **RepeatableVariationPoint**, **ParametricVariationPoint** or **ChoiceVariationPoint**. We divide these variation points into three categories:

- **ChoiceVariationPoint**:
    - **LinkEndSubstitution** (Assigns a new end to a given link)
    - **ObjectSubstitution** (Substitutes an object with another one in the materialized model. It cannot be bound to a VClassifier)
    - **SlotAssignment** (Assigns a new value to a given property of a slot)
    - **FragmentSubstitution** (Substitutes a fragment of the base model with another fragment of the base model. It can be bound to a VClassifier. FragmentSubstitution also inherits from *RepeatableVariationPoint*.)
    - **Existence** (a kind of variation point which indicates that an element of the base model may exist in the materialized model).

  Within the *Existence* category, we distinguish the following concrete variation point:

    - **ObjectExistence** (Indicates that an object of the base model may exist in the materialized model)
    - **SlotValueExistence** (Indicates that a given value of a slot may exist in the materialized model. It permits to replace a given value with an empty value)
    - **LinkExistence** (Indicates that a link of the base model may exist in the materialized model)

- **ParametricVariationPoint**: These variation points are similar to ChoiceVariationPoint except the new value to assign or substitute is stored in a variable
    - **ParametricEndSubstitution** (as LinkEndSubstitution, assigns a new end stored in a variable to a given link )
    - **ParametricSlotAssignment** (as SlotAssignment, assigns a new value stored in a variable to a given property of a slot)

- **ParametricObjectSubstitution** (as ObjectSubstitution, substitutes an object to another one stored in a variable)

- **OpaqueVariationPoint**:
  - **OpaqueVariationPoint** (Expresses variability whose semantics is not defined in CVL)

The following table summarizes the semantics of a variation point taking into account the VSpec (Choice, Variable and VClassifier) that is bound to it. Here, we present both ChoiceVariationPoint and ParametricVariationPoint without distinction.

| | Choice | Variable | Variability classifier |
|---|---|---|---|
| Existence | If the choice is decided positively then the element will be present in the materialized model | Not allowed | Not allowed |
| Slot assignment | If the choice is decided positively then the value associated with the slot assignment will be assigned to the slot | In the resolved system, the value in the variable will be assigned to the slot | Not allowed |
| Substitution | If the choice is decided positively then the substitution will be performed during materialization | The value in the variable will be the replacement | The substitution will be performed once for each instance of the classifier (currently only for fragment substitution) |

We present the semantics of each of the variation points below.

For each of these variation point primitives, we present an example with the printer example and the pre- and post-conditions to use it can be found through the CVL Manual. We also provide an example of code for this variation point in Kermeta in the annex.

Please refer to the manual to see the pre- and post-conditions for the CVL elements previously presented.

#### 10.1.1.1.3.3 Variability encapsulation unit

We have presented the base of the semantics for Variability Abstraction and Realization. We now will present *Configurable Units* which organize elements from Abstraction and Realization. The next section recalls briefly variability configurable unit concepts and defines their semantics.

##### *10.1.1.1.3.3.1 Variability encapsulation unit overview*

As presented in the **Configurable Units** chapter, four main concepts are used:

- **Configurable Unit (CU)** :
  A specific kind of Variation Point that references some base model elements. It may contain other variation points including other Configurable Units; it is a “composite variation point”.

- **CVSpec**:
  VSpec whose resolution requires resolving a set of VSpecs, identified through a variability interface.

- **VInterface**:
  A given collection of VSpec that may be organized as tree.

- **VConfiguration**:
  It resolves a CVSpec by resolving the VSpecs contained in this CVSpec.

  Please see the Figure 38 - VConfiguration resolving a CVSpec, for an example illustrating these concepts.

We can define each of these concepts semantically as follows:

- **VConfiguration** :
  The use of a VConfiguration implies evaluating each variation point linked to the contained VSpecResolutions of the VConfiguration.

- **CVSpec** :
  When a CVSpec is used, we apply the resolution of the VInterface into the associated Configurable Unit
- **VInterface** :
  We iterate on each of the contained VSpec from this VInterface and execute associated Variation Point according to the VConfiguration.

- **Configurable Unit :**
  When a Configurable unit is executed, each of its owned variation points (including inner Configurable Unit) are executed according to the VConfiguration.

*10.1.1.1.3.3.2 CVSpec Derivation*

Instead of having a strong coupling between a VInterface of a given Configurable Unit and another VInterface of an inner Configurable Unit, CVSpec Derivation permits to map these two interfaces with less coupling.

A CVSpec Derivation permits to deduce resolution for inner units from those of outer units and reduce explicit resolutions by user. It resolves a part of inner units.

*10.1.1.1.3.3.3 Configurable Unit Usage*

This variation point permits to configure a given Configurable Unit according to its usage (for example “mainPrinter” or “backup printer”. The aim is to modify the base model by cloning the associated configurable unit and to configure it appropriately.

We have presented the semantics of Variability Abstraction, Variability Realization and Configurable Units. The next section presents some examples of Variation Points of Variability Realization and Configurable Units.

## 10.2 Example presentation

On this part, we focus on expressing variations on the Scanner of the printer running example (see **7 Common Variability Language (CVL) Overview**) to complete previous presentations on **Variability Realization (section 7.4)** and **Configurable Units (section 7.5)** chapters.

### 10.2.1 Choice Variation Point and Parametric Variation Point

The following figure reminds the base model used in this chapter. We only choose to add an attribute for the Scanner *companyName,* which specifies what company provides the scanner.

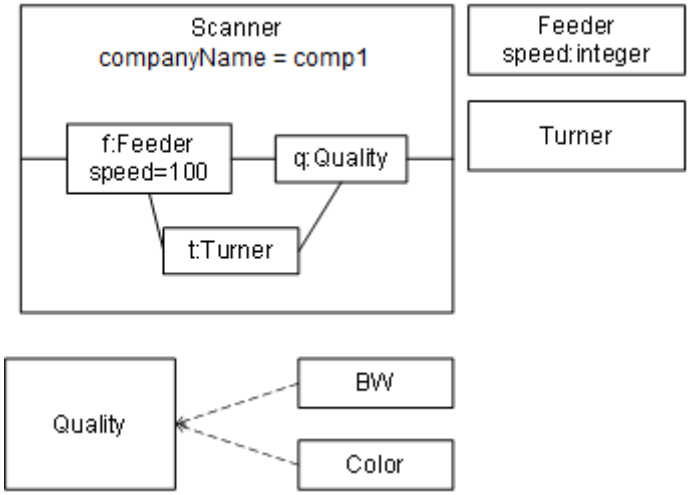



**Figure 60: Scanner base model**

VSpec tree used to express variability on scanner is presented on Figure 61: VSpec tree for scanner base model.

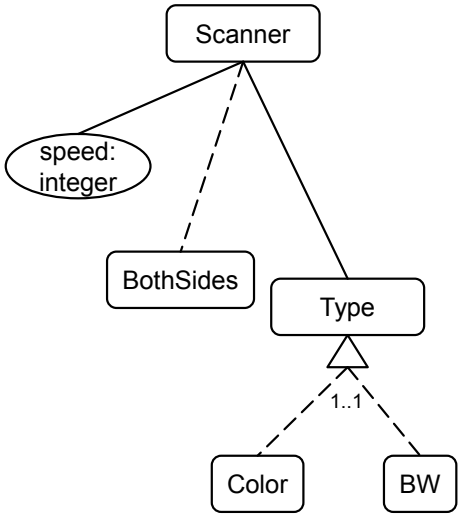



**Figure 61: VSpec tree for scanner base model**

Let us remind the Variability Realization example.

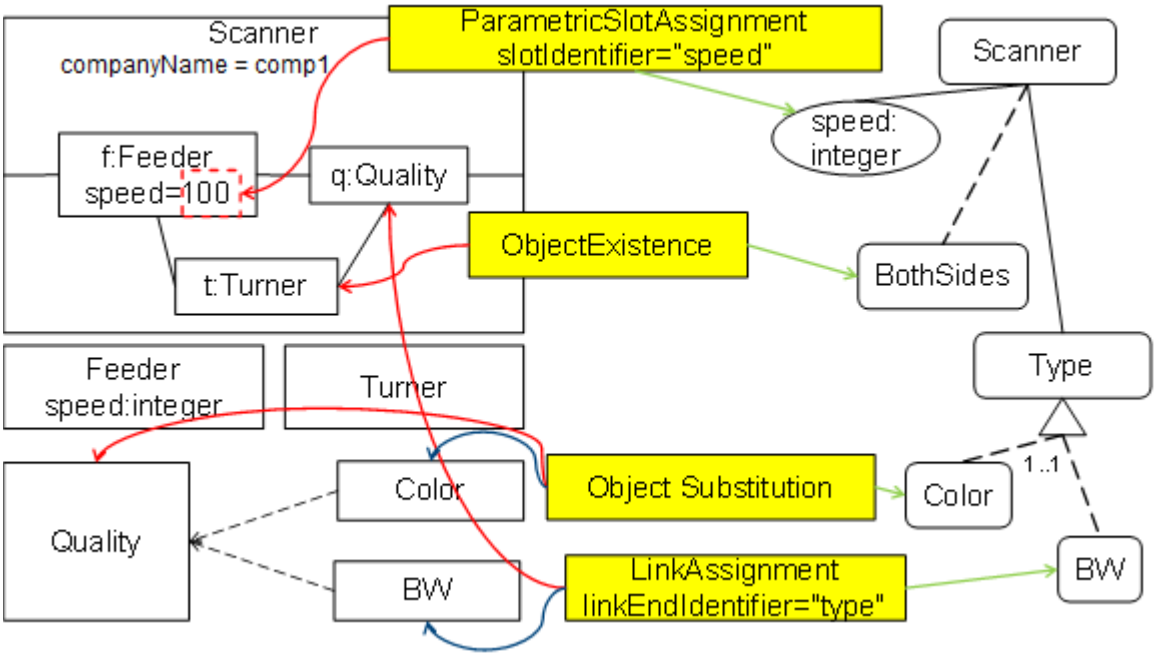



**Figure 62: VSpec tree and Realization on base model**

On the next section, we present in more details what happens during the dynamic semantics execution.

#### 10.2.1.1 LinkEndSubstitution

In the base model represented in the **Figure 63: Update end reference to b : BW**, we change end reference from color to BW.

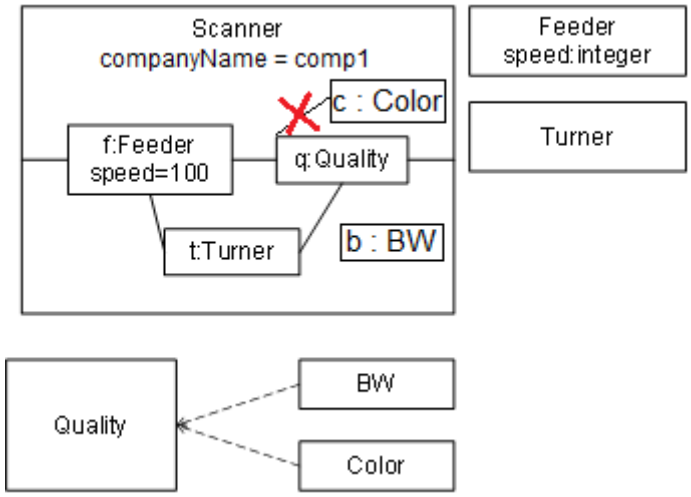



**Figure 63: Update end reference to b : BW**

The result obtained after execution of a LinkEndSubstitution is presented on the **Figure 64: Obtained resolved model :**

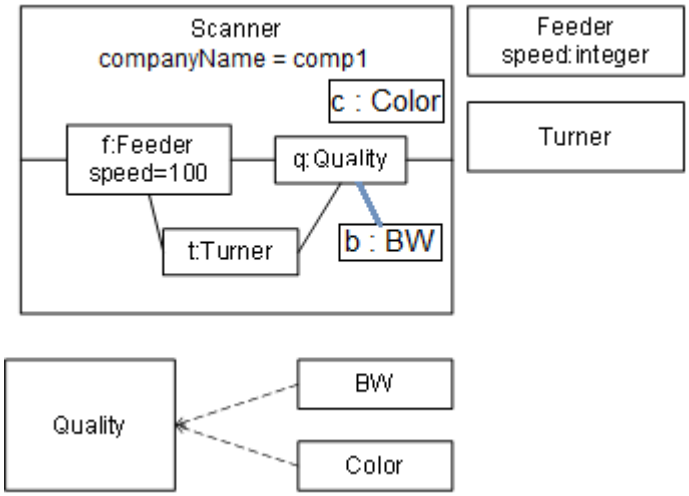



**Figure 64: Obtained resolved model**

#### 10.2.1.1 ObjectSubstitution

In the model presented in **Figure 65: Substitutes c : Color to q : Quality** we substitutes c : Color ( *replacement*) to q : Quality *(placement*).

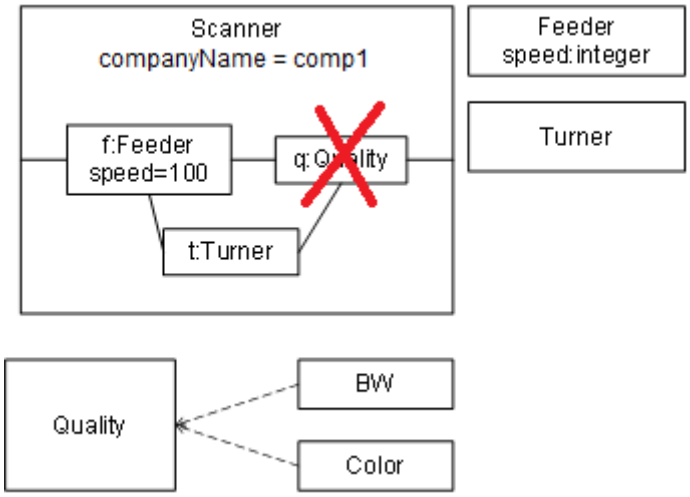



**Figure 65: Substitutes c : Color to q : Quality**

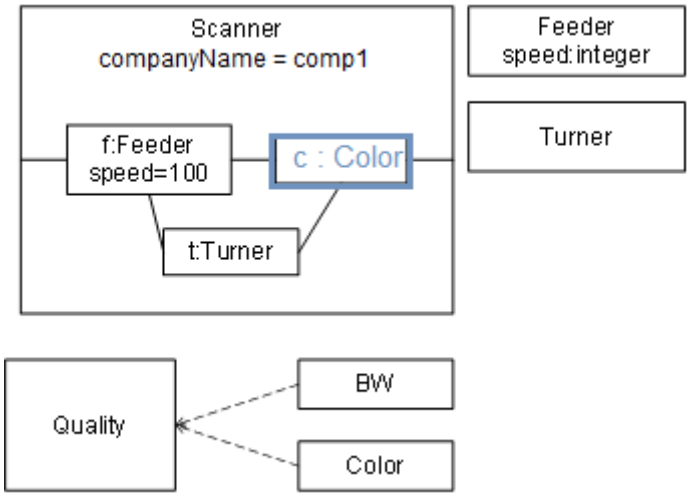



**Figure 66: Obtained resolved model**

#### 10.2.1.1 SlotAssignment

We replace the companyName value comp1 by comp2 in the base model from **Figure 67: Replace comp1 by comp2**.

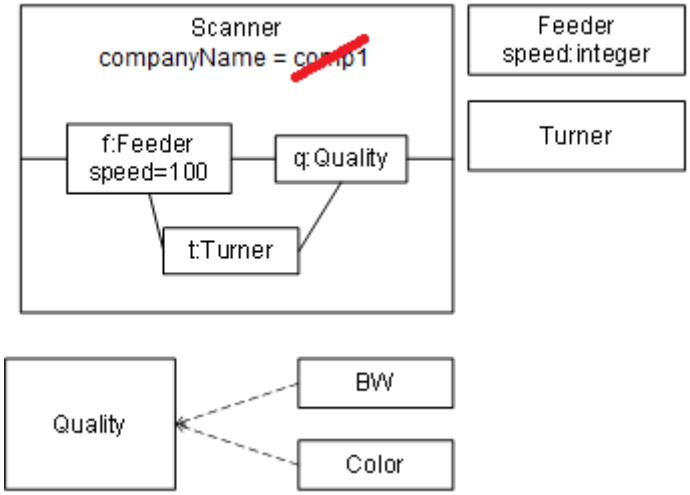



**Figure 67: Replace comp1 by comp2**

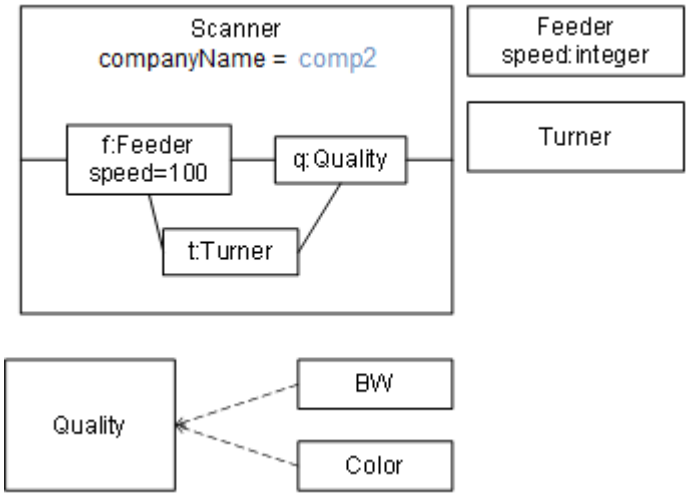



**Figure 68: Obtained resolved model**

#### 10.2.1.1 ObjectExistence

In our case we suppose that the Turner object from the model on figure does not exist.

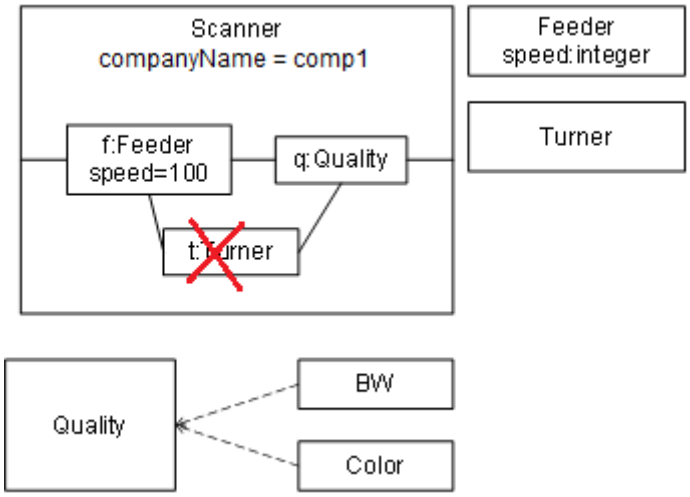



**Figure 69: t: Turner object does not exist**

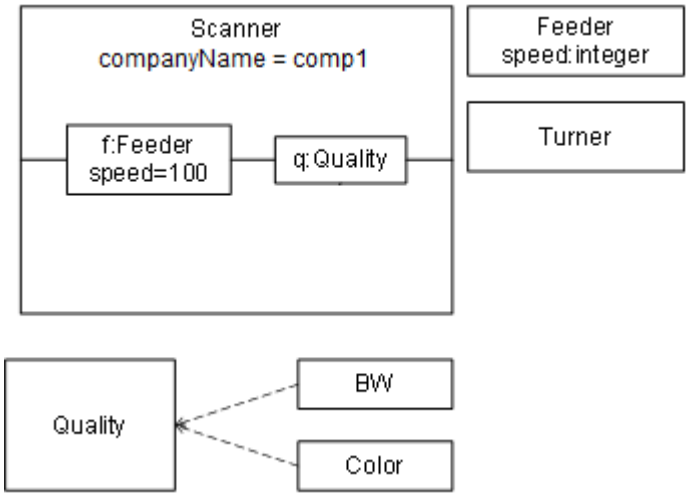



**Figure 70: Obtained resolved model**

#### 10.2.1.1 SlotValueExistence

We suppress the value of the *companyName* attribute from the printer model.

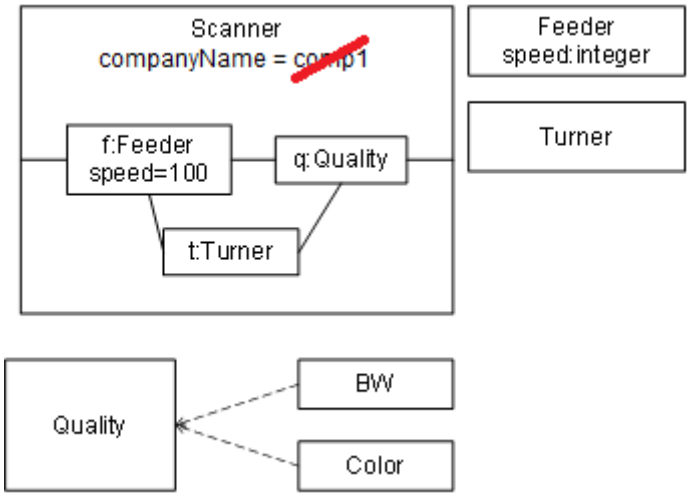



**Figure 71: value of companyName attribute does not exist**

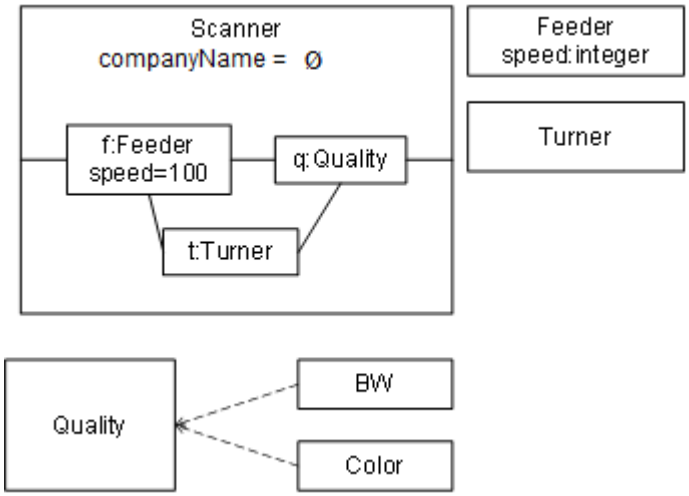



**Figure 72: Obtained resolved model**

#### 10.2.1.1 LinkExistence

Let us suppose that we have a quality q that references the type Black and White bw as in the figure Figure 73: base model for link existence.

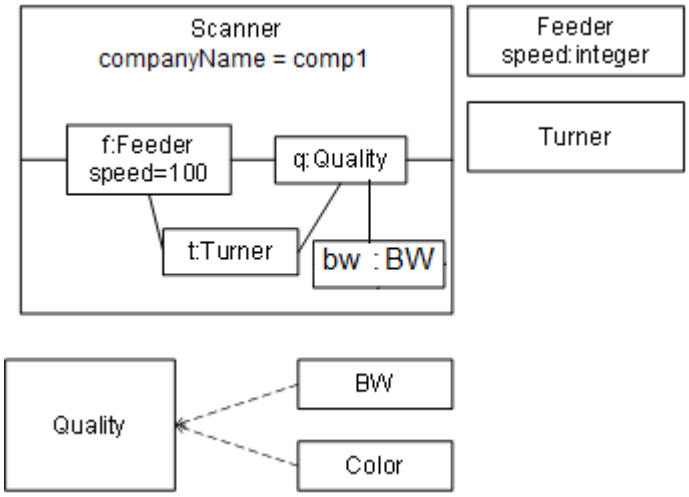



**Figure 73: base model for link existence**

We want to suppress this link type.

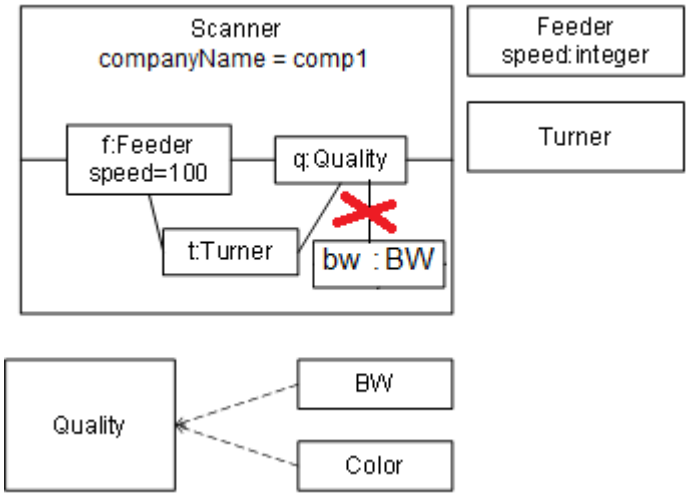



**Figure 74: Suppress type link**

As a result, we obtain a resolved model as in the **Figure 75: Obtained resolved model**.

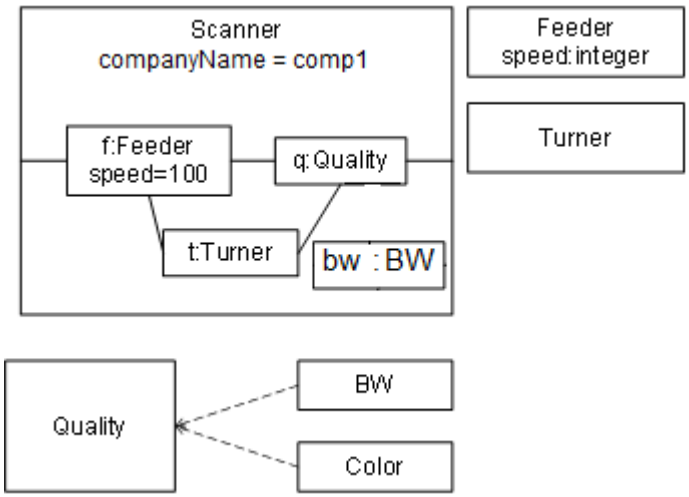



**Figure 75: Obtained resolved model**

#### 10.2.1.1 ParametricLinkEndSubstitution, ParametricSlotAssignment, ParametricObjectSubstitution

These examples are similar respectively to LinkEndSubstitution, SlotAssignment and ObjectSubstitution, except that the value to assign or substitute is stored into a *variable* in the *resolution model*.

## 10.2.2 Repeatable Variation Point

This section deals with the Fragment Substitution Variation Point already presented in the **Variability Realization** chapter (see section **7.4**). We take the same example as presented before, one example with only one fragment substitution and another one with two fragment substitutions.

#### 10.2.2.1 One Fragment Substitution execution

We suppose we have the base model presented on **Figure 76: Base model and Placement for Fragment Substitution**.

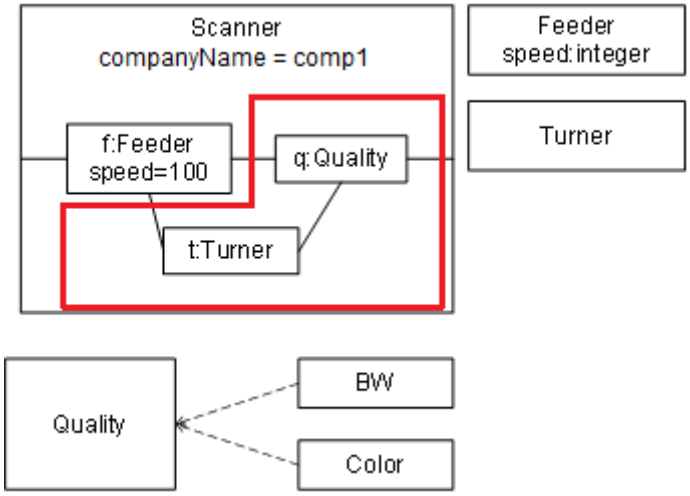



**Figure 76: Base model and Placement for Fragment Substitution**

A placement fragment that contains a quality and a turner element is outlined on this figure.

The figure **Figure 77: Replacement fragment** presents a replacement for these elements a color and a turner t2 element.

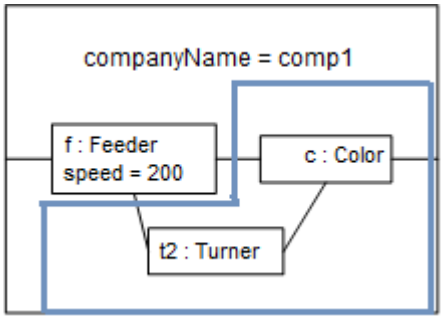



**Figure 77: Replacement fragment**

The Fragment Substitution variation point updates references outside the placement to the replacement object before removing placement object.

So, we obtain the resolved model presented in the **Figure 78: Obtained resolved model**:

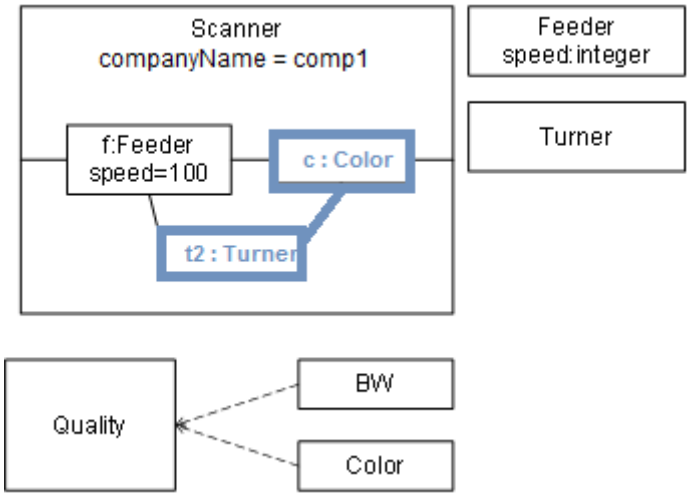



**Figure 78: Obtained resolved model**

#### 10.2.2.1 Two Fragment Substitution execution

On this part we retrieve the example presented in the part Repeatable Variation Point.
The **Figure 79: Base model** presents the base model used for this Fragment Substitution.

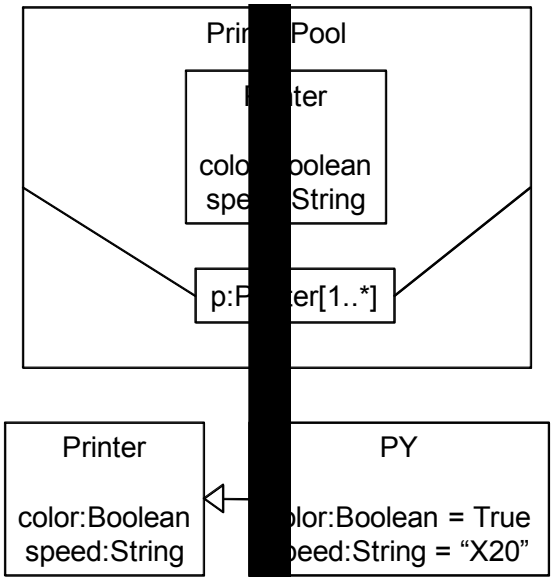



**Figure 79: Base model**

The following VSpec tree represents a printer pool that may have several printers with a choice of speeds between X20 and X22.

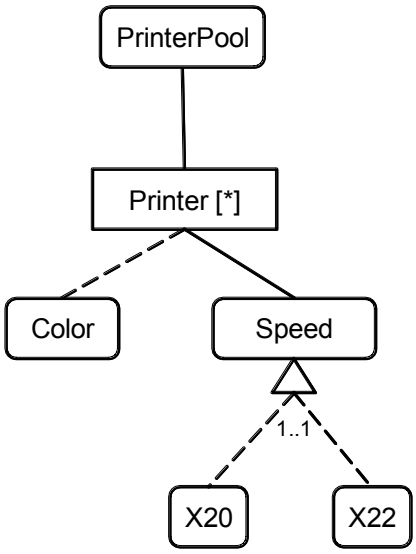



**Figure 80: VSpec tree of the Printer pool**

We choose to add two printers, one with X20 and another one with X22 as speed.

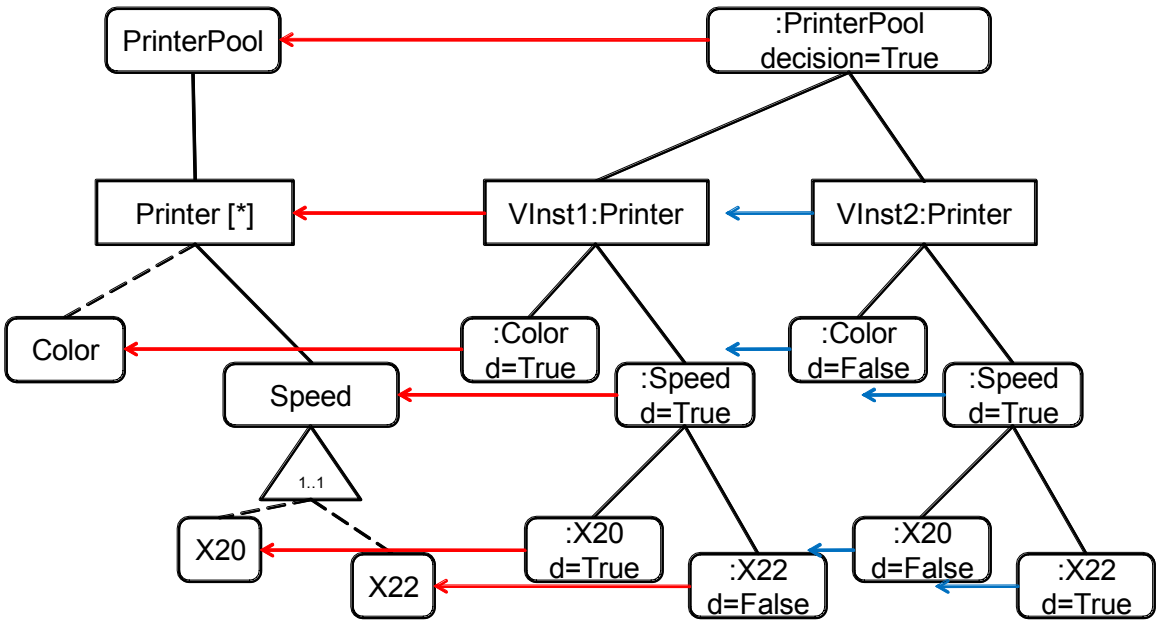



**Figure 81: Resolution model**

For this base model, we apply a Fragment Substitution that adds the replacement two times in the base model and then two Slot Assignment are applied for these replacements added in the base model.

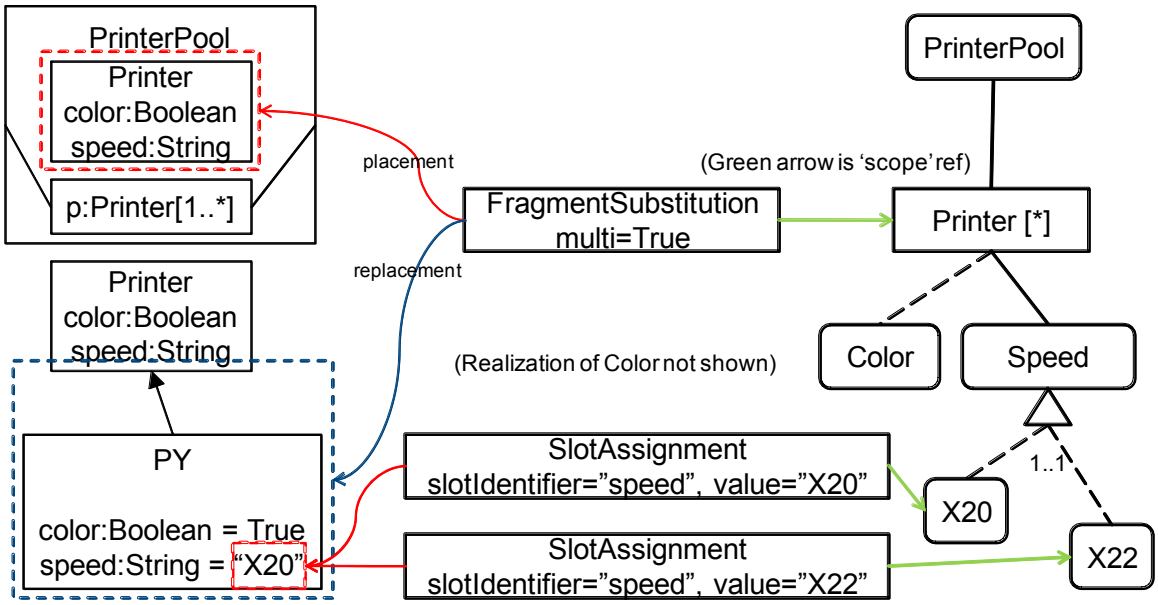



**Figure 82: realization - Variation Point**

We obtain the resolved model presented on **Figure 83: Obtained resolved model**.

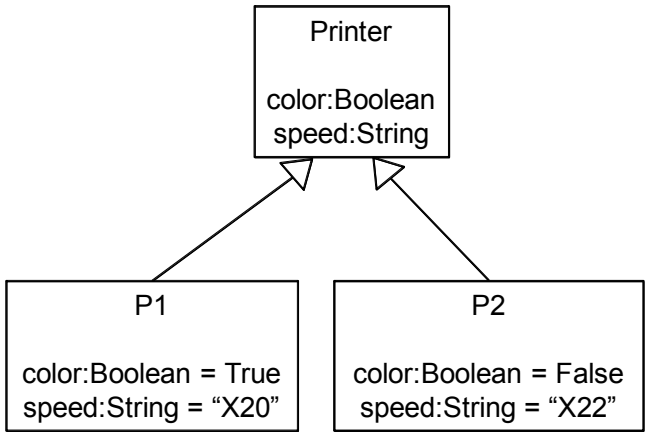



**Figure 83: Obtained resolved model**

## 10.2.3 Opaque Variation Point

By definition, it is an executable, domain-specific variation point whose semantics is not defined by CVL. So, we do not present an example for this variation point. Please refer to the **Opaque Variation Point** chapter for more information about it.

## 10.2.4 Variability encapsulation unit example

### 10.2.4.1 Variability Encapsulation unit global example

We present an example from the chapter Variability Encapsulation, an office package has a printer package and a scanner package.

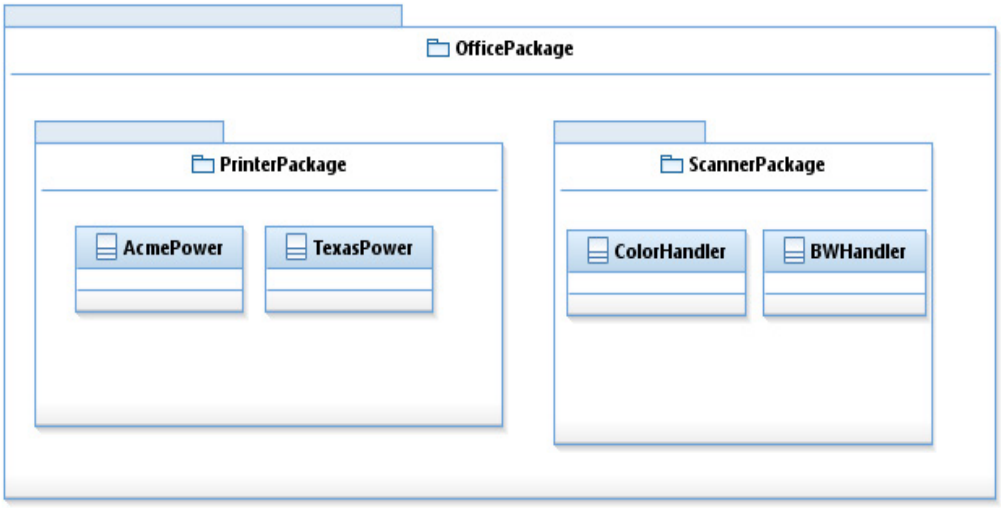



**Figure 84: Base mode for Configurable Unit Example**

We use a Configurable Unit to express variability on this base model.

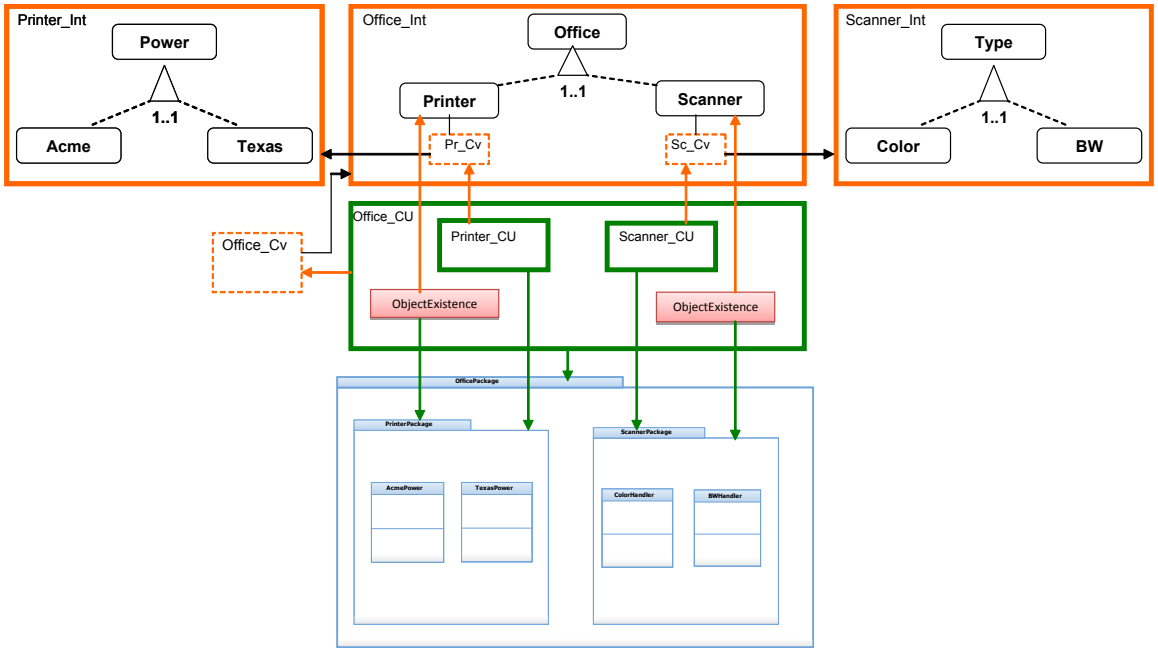



**Figure 85: Configurable Unit definition**

The two Object Existence determine whether respectively the PrinterPackage or the Scanner package exists.
The inner Printer_CU Configurable Unit permits to select between Acme and Texas power whereas the inner Configurable Unit Scanner_CU permits to select between Color and Black and White.

In this configuration, we choose the printer package with the acme power.

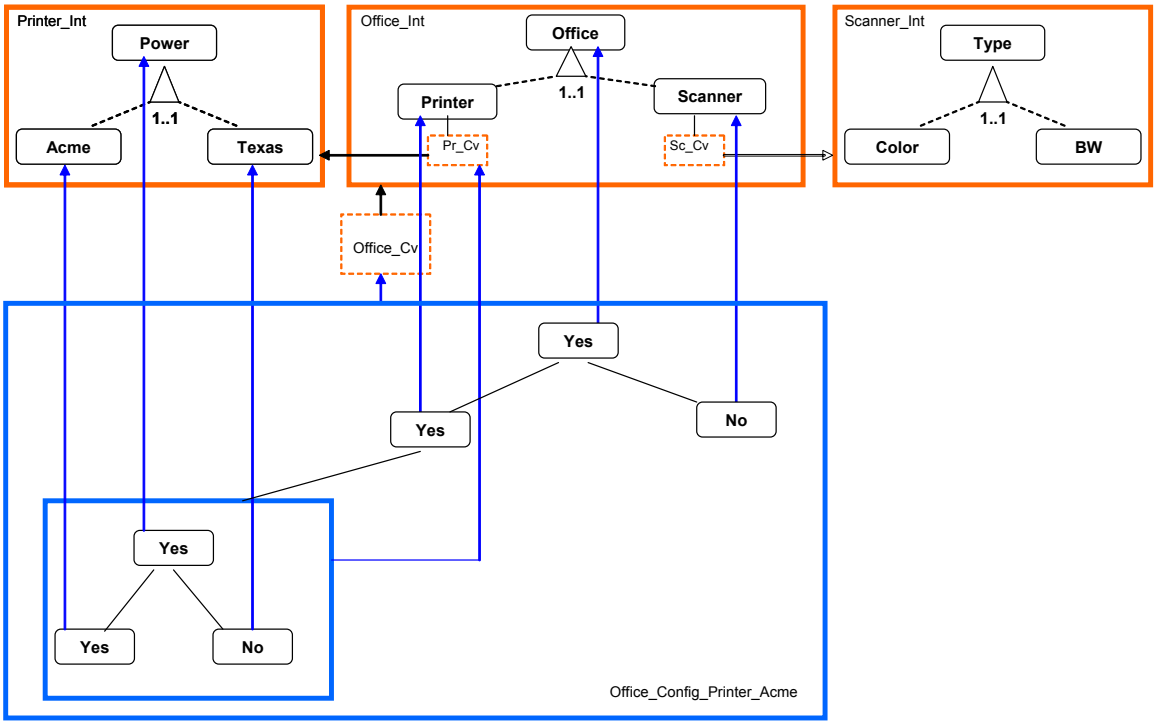



**Figure 86: Configurable Unit Configuration**

This configuration drives to execute the two Object Existence and the two configurable units that are executed according to the decision from the appropriate VSpecResolution. In our case, it drives to suppress unselected package ScannerPackage and Acme Power.

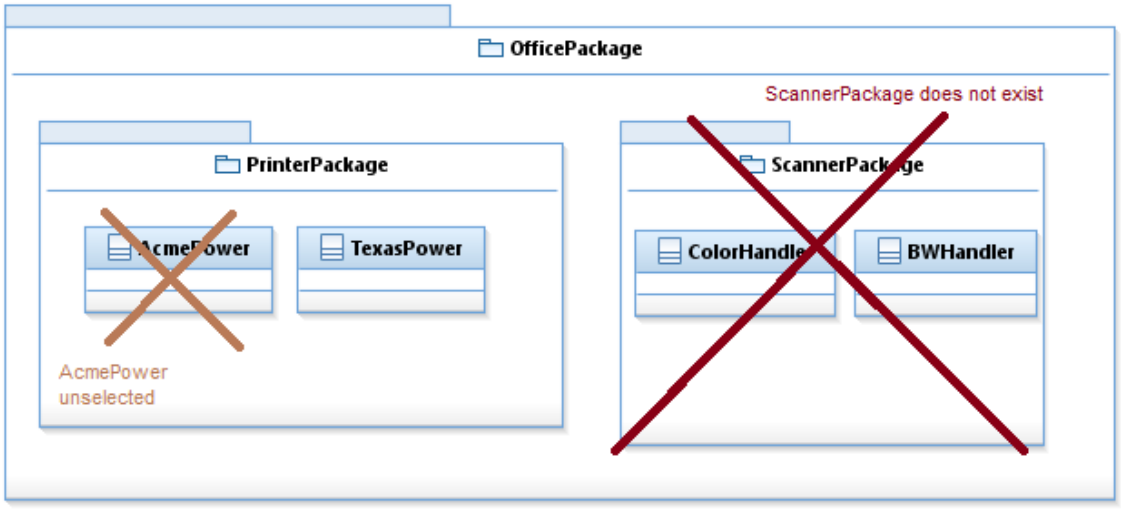



**Figure 87: Configurable Unit Execution on Base Model**

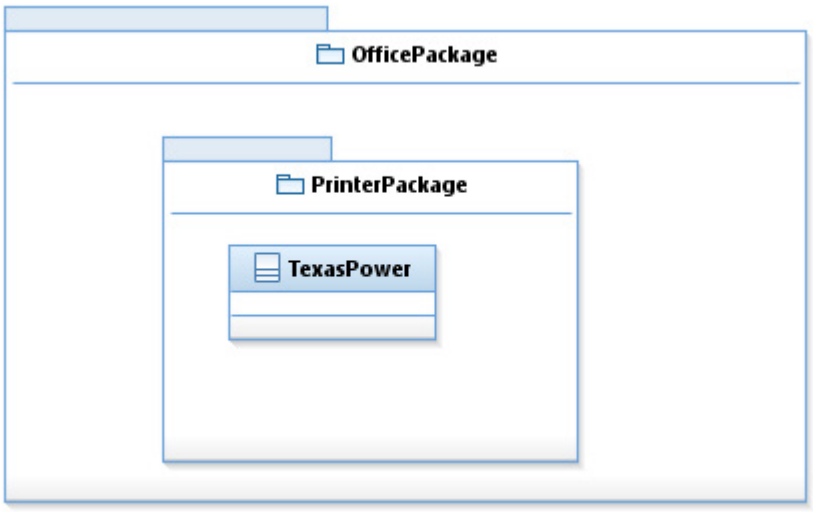



**Figure 88: Obtained base model**

#### 10.2.4.1 CVSpec Derivation

We can add a CVSpec Derivation to the previous example and resolve it with the same result as presented previously.

#### 10.2.4.2 Configurable Unit Usage

We can define two Configurable Unit Usage on the previous example for example ColorUsage and BWUsage and configure respectively the Scanner_CU according to its usages. If we select ColorUsage and Scanner, we obtain a Scanner with color and if we select BWUsage and Scanner we obtain a Scanner with Black and white. For an example on the configurable unit usage, please see Section 7.5.6 of this document.

This chapter has presented dynamic semantics for Variability Abstraction, Variability Realization, and Configurable Units.

# MANDATORY CVL *NORMATIVE* SPECIFICATIONS

# 11 CVL Metamodel

This chapter contains the diagrams of the metamodel constructed in RSA 8.
This chapter gives the details of the metamodel and semantics for every metaclass.

## 11.1 Abstract Syntax Diagrams

### 11.1.1 Variability Abstraction

#### 11.1.1.1 VSpec and Resolution

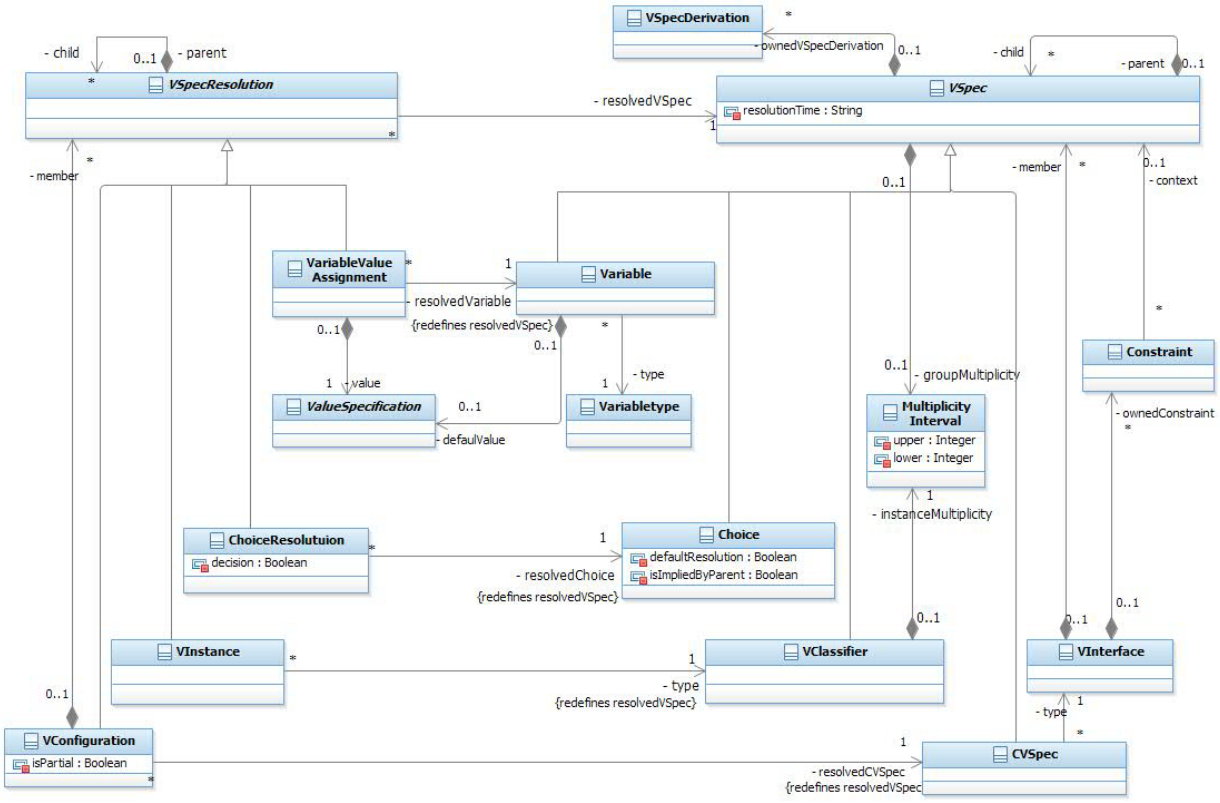



**Figure 89 VSpecs and corresponding VSpecResolutions**

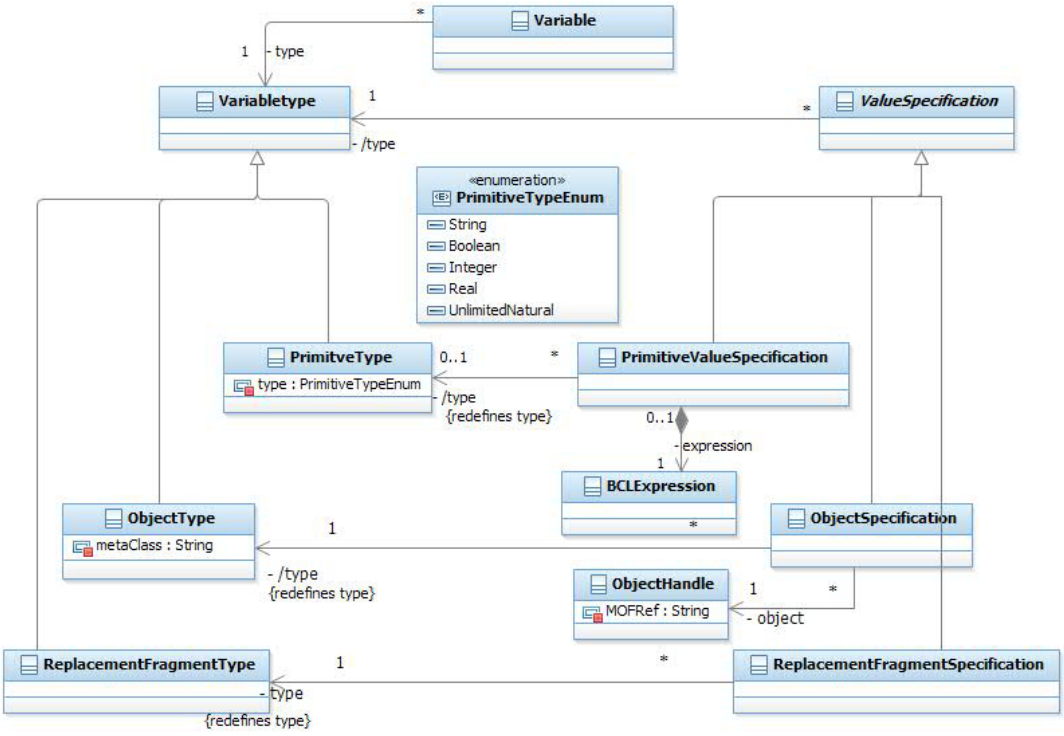



**Figure 90 Variable and associated ValueSpecification**

#### 11.1.1.2 Constraints

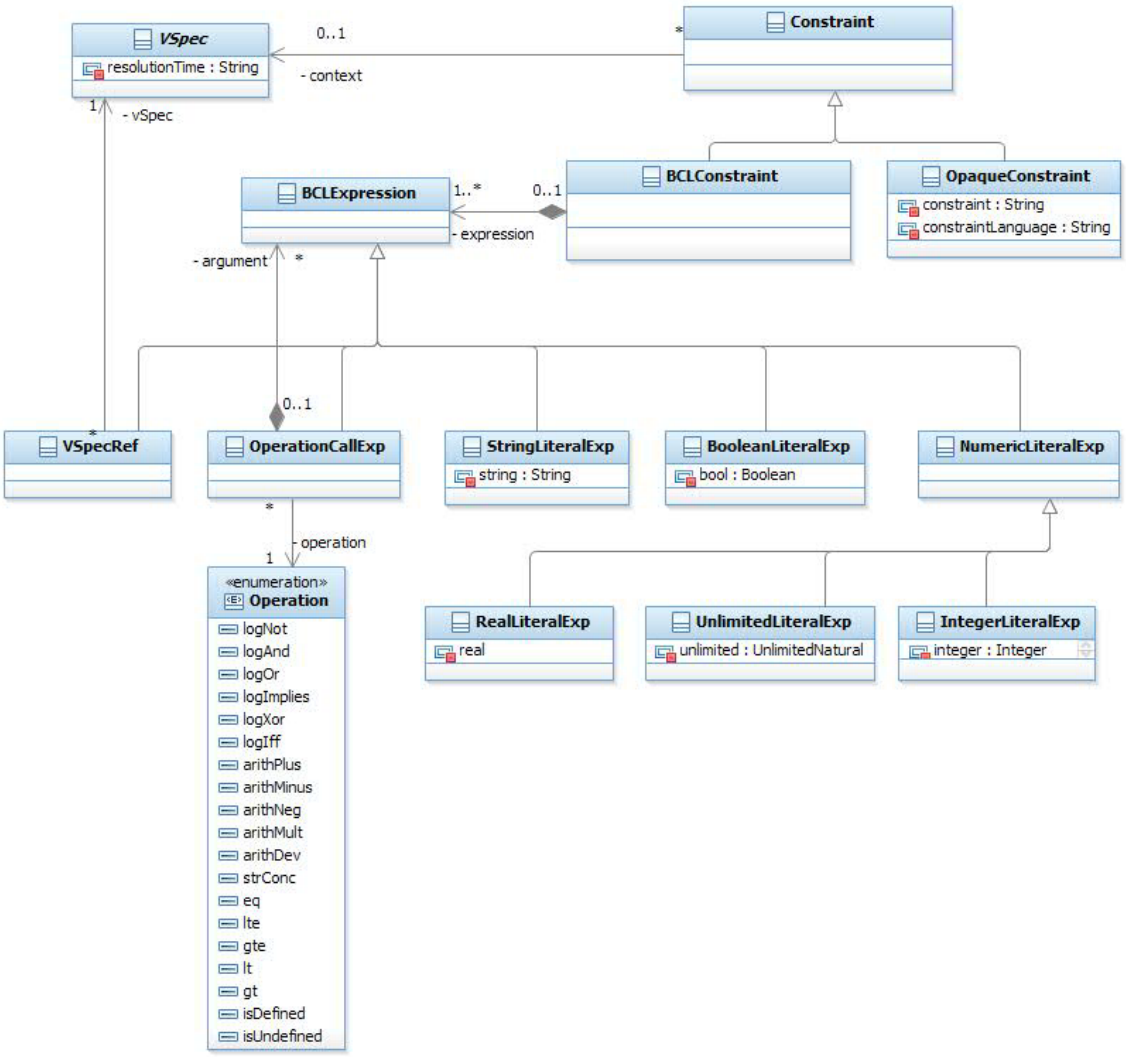



**Figure 91 Constraints (Basic Constraint Language and Opaque Constraint)**

#### 11.1.1.3 VPackage

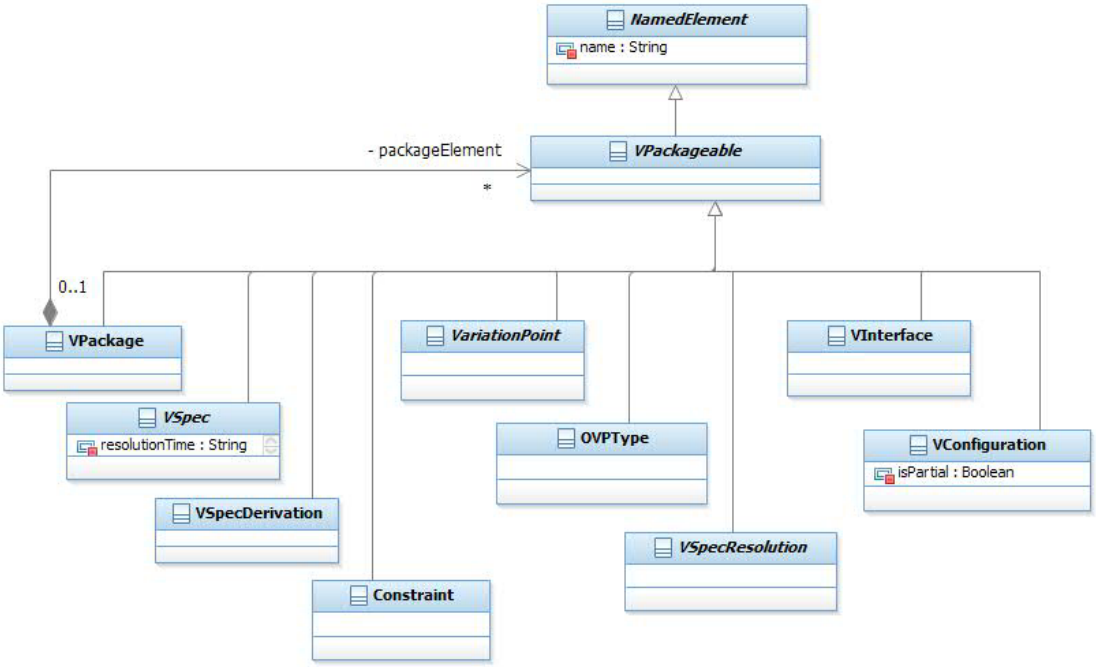



**Figure 92 VPackage and the VPackageable**

## 11.1.2 Variability Realization

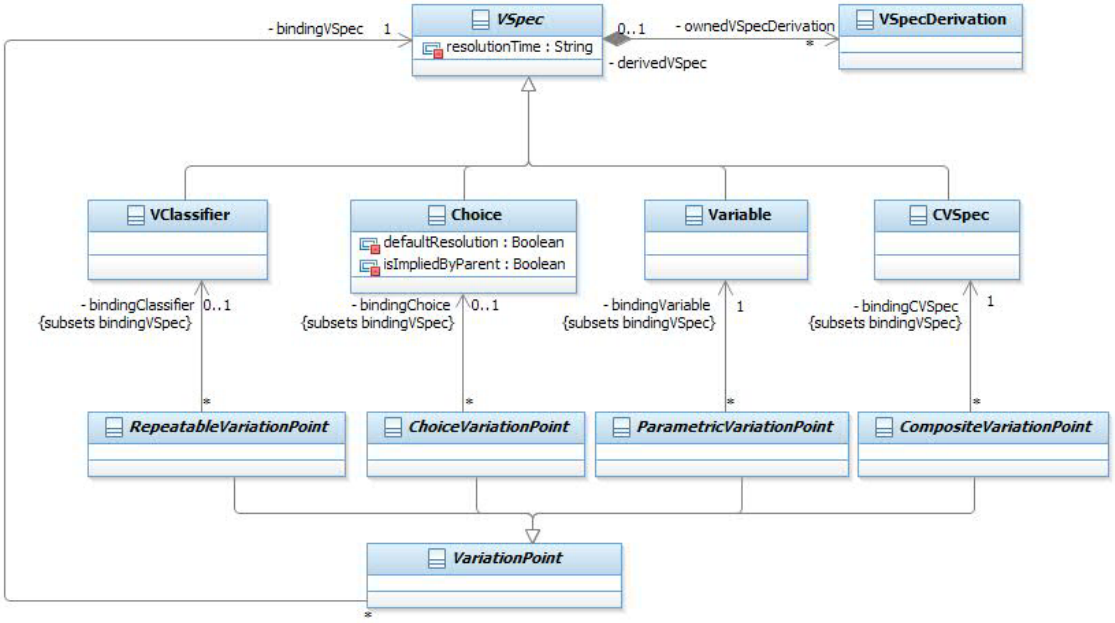



**Figure 93 Binding between Variation Points and VSpecs**

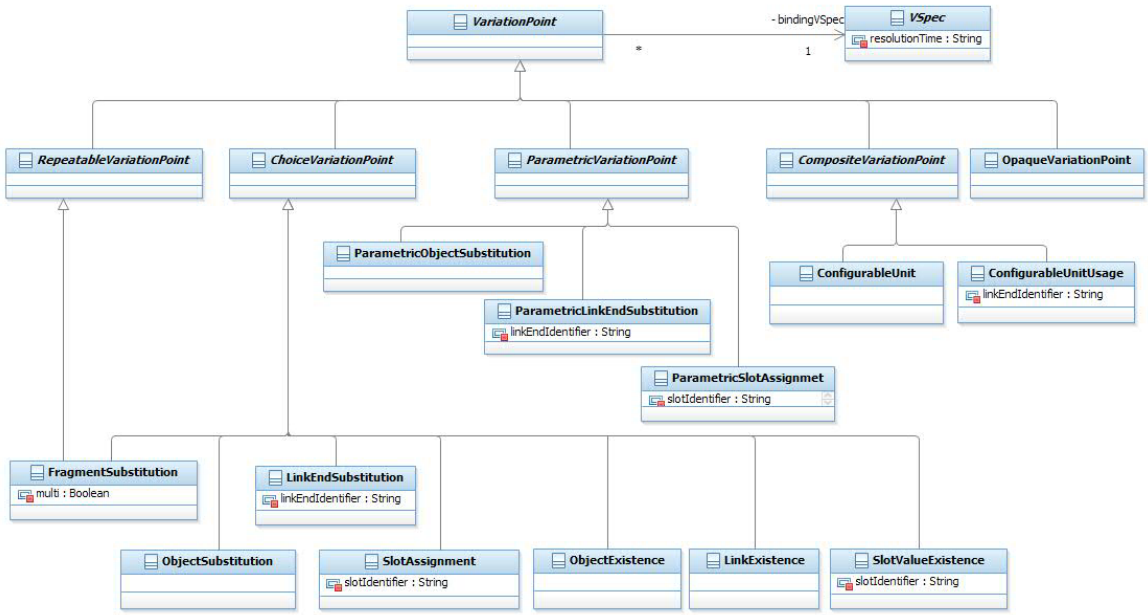



**Figure 94 Variation Point Taxonomy**

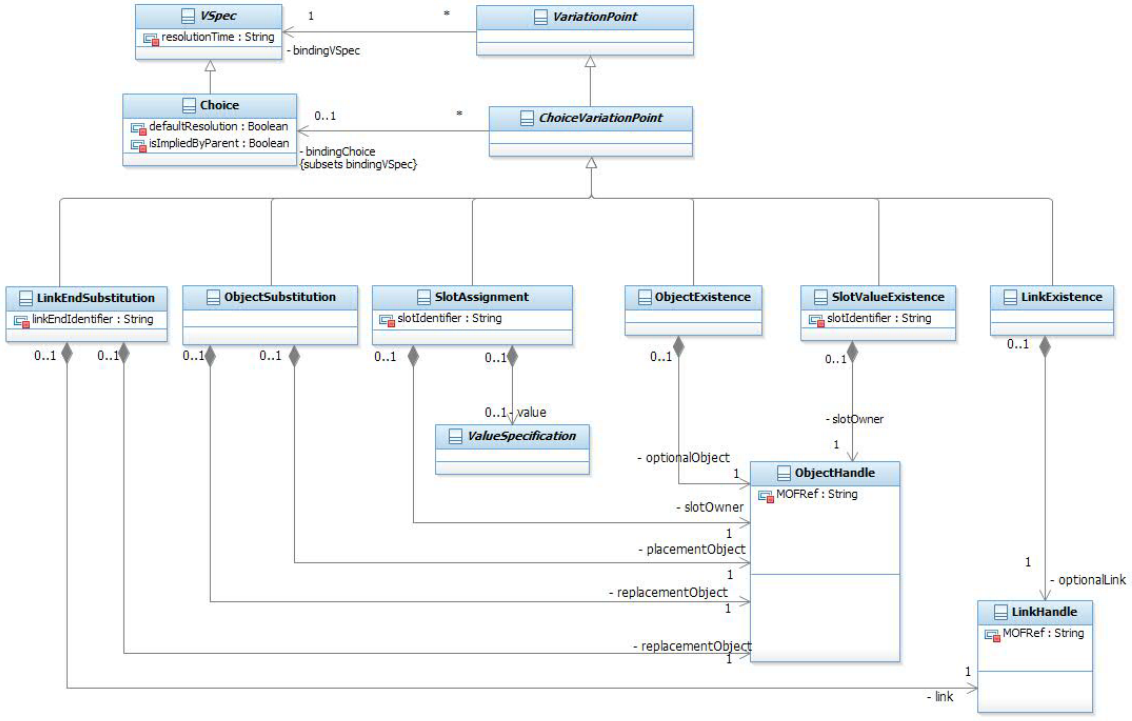



**Figure 95 Choice Variation Point details**

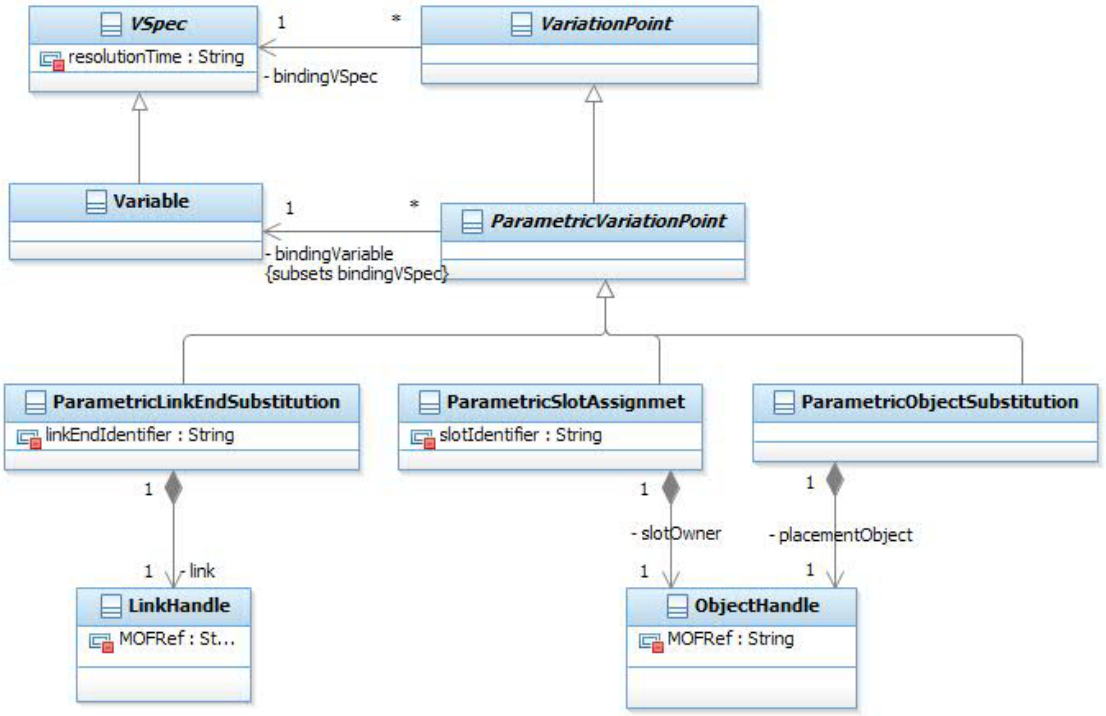



**Figure 96 Parametric Variation Point details**

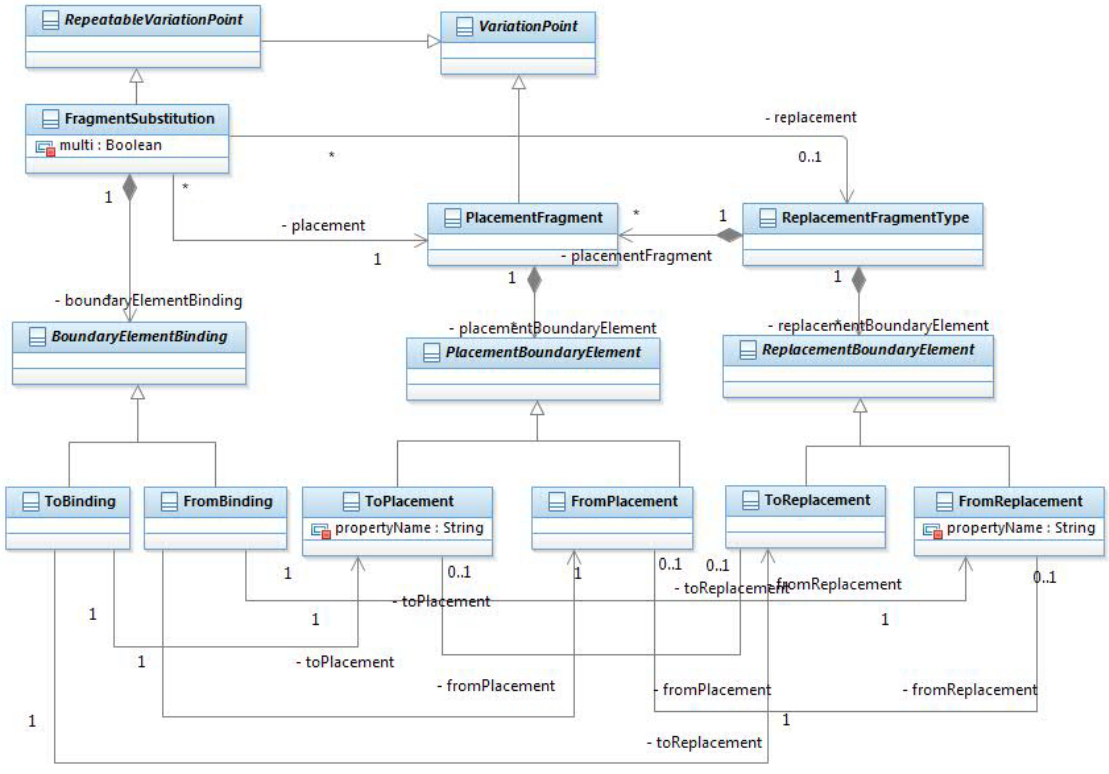



**Figure 97 Fragment Substitution details**

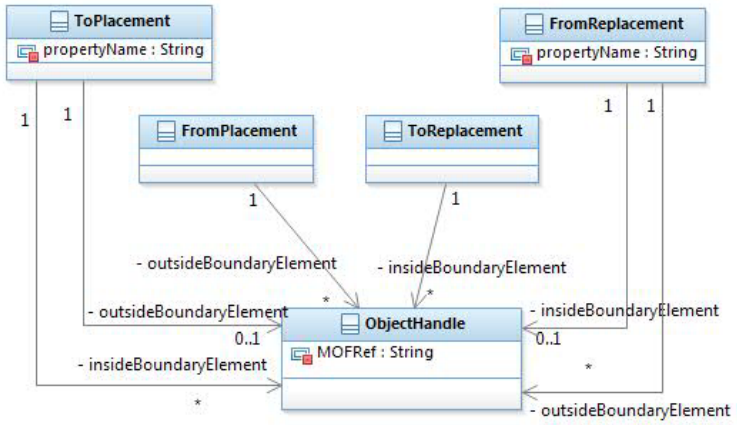



**Figure 98 Boundary Elements for fragment definition**

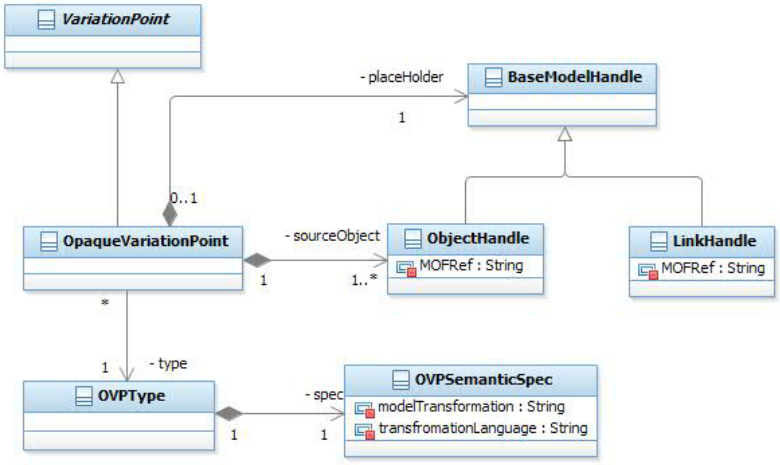



**Figure 99 Opaque Variation Point details**

### 11.1.3 Configurable Unit and VInterface

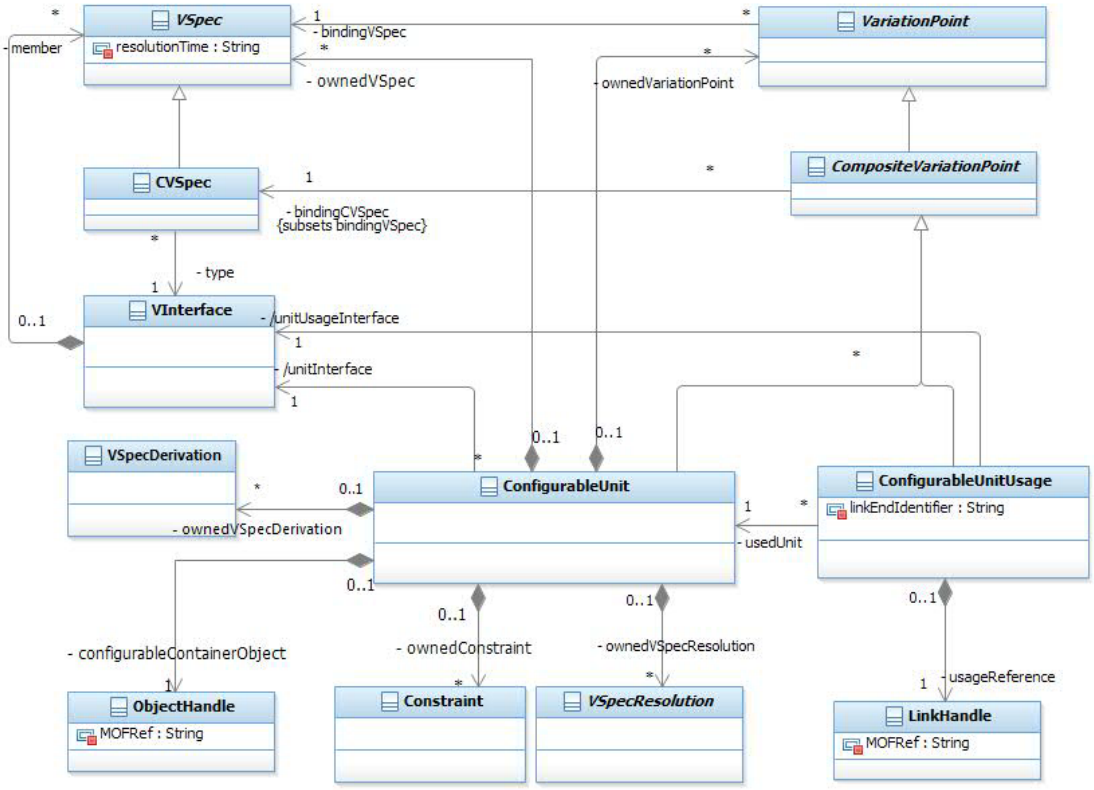



**Figure 100 Configurable Units**

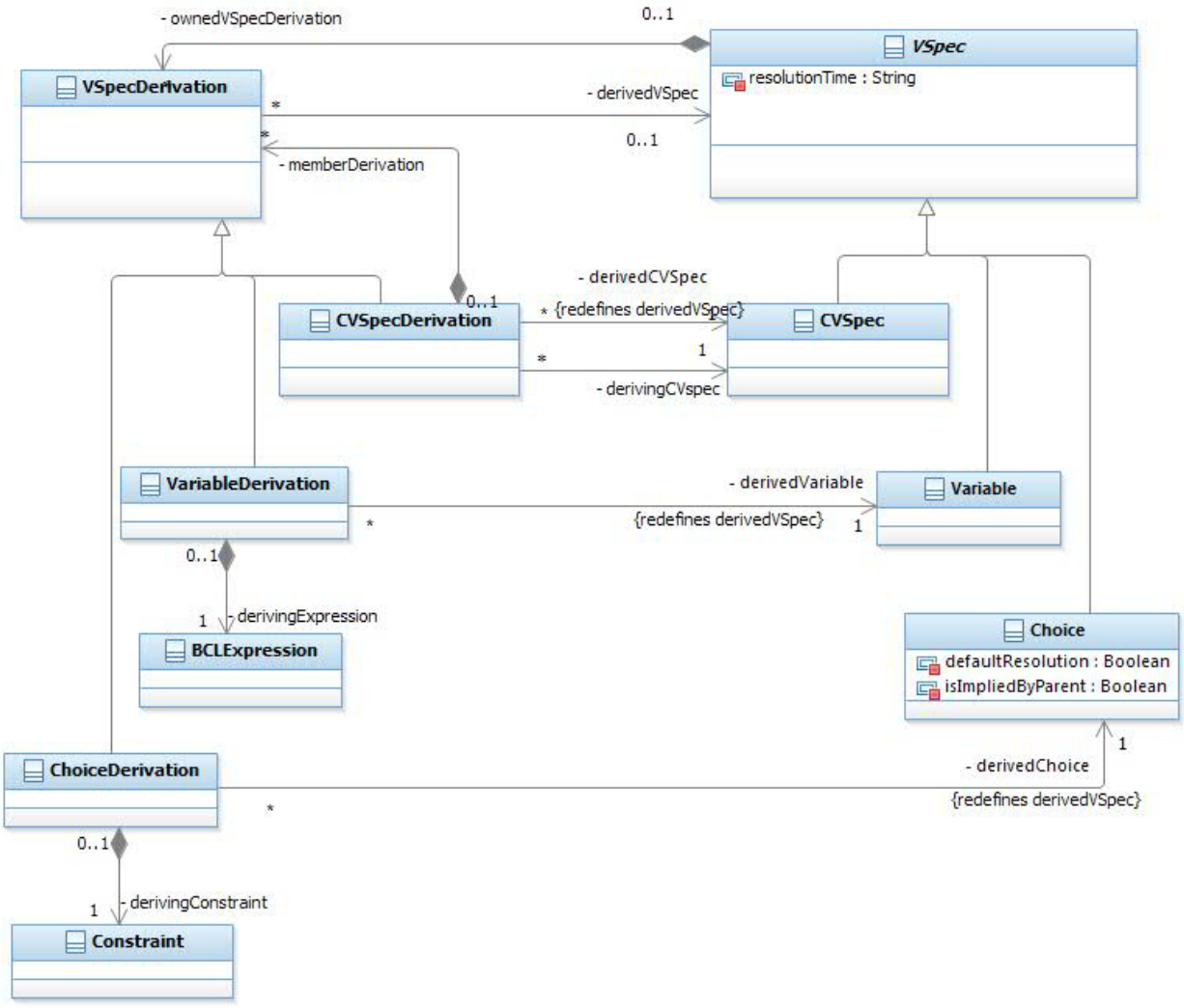



**Figure 101 VSpec derivation**

# 11.2 CVL Metamodel Manual

## 11.2.1 BaseModelHandle

### 11.2.1.1 Description

Represents the commonalities of ObjectHandle and LinkHandle as the means to relate to objects of the base model.

## 11.2.2 BCLConstraint

### 11.2.2.1 Description

Top class of a constraint. Contains basic constraint language expressions. A constraint can have local context, which is a VSpec (we refer to it as the context VSpec), or it can have a global context (i.e., the

constraint is global within its enclosing package or configurable unit, and it has no context VSpec). The context determines the VSpec(s) for which the constraint must hold.

#### 11.2.2.2 Generalizations

Constraint

#### 11.2.2.3 Associations

- expression : BCLExpression [1..*]
  The constraining expression of this BCLConstraint.

### 11.2.3 BCLExpression

#### 11.2.3.1 Description

A generic class for expressions available in the basic constraint language.

### 11.2.4 BooleanLiteralExp

#### 11.2.4.1 Description

A literal expression that represents Boolean values ('true' or 'false').

#### 11.2.4.2 Generalizations

BCLExpression

#### 11.2.4.3 Attributes

- bool : Boolean [1..1]
  The boolean literal value of this expression.

### 11.2.5 BoundaryElementBinding

#### 11.2.5.1 Description

Specifies the binding between the boundary elements of the placement fragment and the replacement fragment.

### 11.2.6 Choice

#### 11.2.6.1 Description

A choice is VSpec whose resolution requires a yes/no decision (True/False). When a variation point is bound to a choice, the decision resolving that choice determines whether or not the variation point will be applied during materialization.

#### 11.2.6.2 Generalizations

VSpec

#### 11.2.6.3 Attributes

- defaultResolution : Boolean [0..1]
  The default resolution of this choice.
- isImpliedByParent : Boolean [1..1]
  When True then resolving the parent VSpec positively implies deciding this choice positively. A VSpec resolution is positive if it is a choice decided positively, or any classifier instantiation, or any value assignment to a variable.<br><br>For a root choice, True implies it must be resolved positively.

#### 11.2.6.4 Semantics

Invariant : If a choice is implied by parent, it must have a parent.
OCL :
-- Choice
-- If a choice is implied by parent, it must have a parent.
context Choice :
inv isImpliedByParentsImpliesAParent :
self.isImpliedByParent implies VSpec.allInstances()->exists(vSpec | vSpec.childVSpec->includes(self))

### 11.2.7 ChoiceDerivation

#### 11.2.7.1 Description

A choice derivation is a specification how to derive the decision for a particular choice from resolutions of other VSpecs using a constraint.

#### 11.2.7.2 Generalizations

VSpecDerivation

#### 11.2.7.3 Associations

- derivedChoice : Choice [1..1]
  The derived choice of this ChoiceDerivation.
- derivingConstraint : Constraint [1..1]
  The deriving constraint.

### 11.2.8 ChoiceResolutuion

#### 11.2.8.1 Description

A ChoiceResolution is a
VSpecResolution which resolves a single choice by deciding it positively (True) or negatively (False).

#### 11.2.8.2 Generalizations

VSpecResolution

#### 11.2.8.3 Attributes

- decision : Boolean [1..1]
  The yes/no decision resolving the choice.

#### 11.2.8.4 Associations

- resolvedChoice : Choice [1..1]
  The resolved choice of this resolution.

#### 11.2.8.5 Semantics

Invariant
Invariant : If a choice is selected, the number of selected children must correspond to the multiplicity interval of the resolvedChoice
OCL :

```
-- ChoiceResolution
-- If a choice is selected, the number of selected children must correspond to the multiplicity interval of the resolvedChoice
context ChoiceResolutuion :
inv selectedChildrenMustCorrespondsToMultiplicityInterval :
if self.resolvedChoice.groupMultiplicity->isEmpty()
then
(self.resolvedChoice.groupMultiplicity.upper <> (-1)
and self.resolvedChoice.groupMultiplicity.lower <= self.childResolution->select (choiceRes |
choiceRes.oclAsType(ChoiceResolutuion).decision)->size()
and self.childResolution->select (choiceRes | choiceRes.oclAsType(ChoiceResolutuion).decision)->size()
>= self.resolvedChoice.groupMultiplicity.upper
)
or
(self.resolvedChoice.groupMultiplicity.upper == (-1)
and self.resolvedChoice.groupMultiplicity.lower <= self.childResolution->select (choiceRes |
choiceRes.oclAsType(ChoiceResolutuion).decision)->size()
)
else
-- no choice must be selected
self.childResolution->select (choiceRes | choiceRes.oclAsType(ChoiceResolutuion).decision)->isEmpty()
endif
```

Dynamic semantics
Pre and post condition
Pre-condition :
None
Post-condition:
After a Choice Resolution has been executed, the resolvedChoice is constained in the selected VSpecs set if the boolean decision is set to true, otherwise, the resolvedChoice is contained in the unselected VSpec set.
OCL :

```
-- ChoiceResolution
context ChoiceResolutuion::eval(ctx : CVLExecutionContext):
pre :
post :
self.decision implies
ctx.selectedVSpecs->includes(self.resolvedChoice)
and not self.decision implies
ctx.unselectedVSpecs->includes(self.resolvedChoice)
```

## 11.2.9 ChoiceVariationPoint

### 11.2.9.1 Description

A choice variation point is a variation point which may be bound to a choice. During materialization the decision resolving the choice determines whether or not the variation point will be applied or not.

### 11.2.9.2 Generalizations

VariationPoint

### 11.2.9.3 Associations

- bindingChoice : Choice [0..1]
  The binding choice.

## 11.2.10 CompositeVariationPoint

### 11.2.10.1 Description

A composite variation point is either a configurable unit or a configurable unit usage. It must be bound to a CVSpec.

### 11.2.10.2 Generalizations

VariationPoint

### 11.2.10.3 Associations

- bindingCVSpec : CVSpec [1..1]
  The binding CVSpec

## 11.2.11 ConfigurableUnit

### 11.2.11.1 Description

A configurable unit (CU) is a kind of variation point which references a base model object, and by doing so indicates the object is a container with inner variability. The latter is specified via contained variation points, so a configurable unit is a kind of variation point which may contain other variation points.A configurable unit represents a base model object which is a reusable component, configurable via an exposed variability interface. The base model object is a container of other base model elements – objects and links – and the CVL configurable unit is a container of variation points defined against those elements, bound to VSpecs. For example, a configurable unit may have associated with it a UML package and contain variation points defined over elements in that package.A CU hides its internals and exposes a variability interface (VInterface) to the external world, through which it may be configured. This is achieved by binding the CU to a CVSpec, whose type is the exposed interface. The configurable unit may then be configured by providing resolutions to the VSpecs in its interface.Conceptually, a configurable unit and the base model object associated with it are one unified entity representing a reusable base model component. So the UML package above may be thought of as being configurable, though technically it is an external CVL element referencing the object through a handle.
CVL configurable units may contain other units, re-iterating the modular structure of the base model. For example, a UML package and a contained class may both be configurable, which means there will be two CVL configurable units, one for the package and one for the class, where the unit for the package contains that for the class. In this way, in addition to facilitating reusable components, configurable units also

facilitate modular design.Configurable units are also packaging/shipping elements and as such may also contain constraints, VSpec derivations, and VSpec resolutions.

#### 11.2.11.2 Generalizations

CompositeVariationPoint

#### 11.2.11.3 Associations

- configurableContainerObject : ObjectHandle [1..1]
  The container object in the base model which is configurable.
- ownedConstraint : Constraint [0..*]
  The constraints wrt the internal VSPecs of the unit.
- ownedVariationPoint : VariationPoint [0..*]
  Internal variation points against the content of the base model object referenced by the unit.
- ownedVSpec : VSpec [0..*]
  Internal VSpecs of the CU
- ownedVSpecDerivation : VSpecDerivation [0..*]
  The owned VSpec derivations.
- ownedVSpecResolution : VSpecResolution [0..*]
  The owned VSpec resolutions.
- unitInterface : VInterface [1..1]
  The interface of the unit. Derived as the type of the CVSpec to which the unit is bound.

#### 11.2.11.4 Semantics

Invariant :
Elements contained in the Configurable Unit associated with a given CVSpec must only point on elements in Interface associated with this given CVSpec
OCL :

```
-- ConfigurableUnit
-- Elements contained in the Configurable Unit associated with a given CVSpec must only point on elements
-- in Interface associated with this given CVSpec
context ConfigurableUnit :
def : isContained ( in vps : VSpec ) : Boolean =
self.bindingCVSpec.vInterface.vSpec->exists(v | v = vps
or
( if not (v.childVSpec->isEmpty())
then
v.childVSpec->exists (child | isContained (child) = true)
endif
)
)
inv consistencyInInterfacePointedByCVSpec :
not (self.ownedVariationPoint->exists (vp |
-- search in the VSpec
( vp.bindingVspec->exists ( vsp | not isContained(vsp) ) )
))
```

## 11.2.12 ConfigurableUnitUsage

### 11.2.12.1 Description

A configurable unit usage is a variation point which facilitates per-usage configuration of a configurable unit. Each usage variation point references the CU of which it is a usage, and also references a base model link-end touching the base model container referenced by the CU referenced by the usage. A usage variation point must be bound to a CSVpec, the resolution of which configures the particular usage. The type of that CVSpec must be the same as the type of the CVSpec to which the configurable unit it uses is bound. During materialization the base model container referenced by the CU (referenced by the usage) is deeply cloned, and the base model link-end identified by the usage is redirected to the clone.

### 11.2.12.2 Generalizations

CompositeVariationPoint

### 11.2.12.3 Attributes

- linkEndIdentifier : String [1..1]
  The identifier of the link end.

### 11.2.12.4 Associations

- unitUsageInterface : VInterface [1..1]
  The interface of the unit usage. Derived as the extension of the binding CVSpec.
- usageReference : LinkHandle [1..1]
  Reference to the usage.
- usedUnit : ConfigurableUnit [1..1]
  The used unit.

## 11.2.13 Constraint

### 11.2.13.1 Description

A constraint specifies restrictions on permissible resolution models.

### 11.2.13.2 Generalizations

VPackageable

### 11.2.13.3 Associations

- context : VSpec [0..1]
  The context of the constraint.

## 11.2.14 CVSpec

### 11.2.14.1 Description

A CVSpec is a VSpec whose resolution requires resolving the VSpecs in its type, which is a VInterface. When a configurable unit is bound to a CVSpec, its resolution determines the transformations to be applied to the internals of the unit during materialization.

### 11.2.14.2 Generalizations

VSpec

### 11.2.14.3 Associations

- type : VInterface [1..1]
  The VInterface whose members need to be resolved in order to resolve the CVSpec.

## 11.2.15 CVSpecDerivation

### 11.2.15.1 Description

A CVSpec derivation is a specification how to derive the resolution for a particular CVSpec from the resolution of another CVSpec

### 11.2.15.2 Generalizations

VSpecDerivation

### 11.2.15.3 Associations

- derivedCVSpec : CVSpec [1..1]
  The derived CVSpec.
- derivingCVspec : CVSpec [1..1]
  The deriving CVSpec.
- memberDerivation : VSpecDerivation [0..*]
  The member derivations.

## 11.2.16 FragmentSubstitution

### 11.2.16.1 Description

Fragment Substitution substitutes a placement fragment of the base model with one or more replacement fragments of the base model.Constraints:The boundary elements define all references going in and out of the placement fragment. The boundary elements fully define all references going in and out of the replacement fragment.Semantics:1. Delete the model elements defined by the PlacementFragment. The placement model elements can be found through FragmentSubstitution.placement's placementBoundaryElements that are of class ToPlacement (using the model element references called insideBoundaryElement) and the transitive closure of all references from these, where the traversal is cut off at any reference that has the same value as any of FragmentSubstitution.placement's PlacementBoundardyElement that are of class FromPlacement (using the model element references called outsideBoundaryElement).

2. For the replacement fragments, copy its content onto the hole made by the deletion of the placement fragment. The placement and replacement boundary elements must correspond. The content model elements can be found through FragmentSubstitution. replacement's ReplacementBoundaryElement that are of type ToReplacement (using the model element references called insideBoundaryElement) and all model elements found through the transitive closure of all references from this set of model elements, where the traversal is cut off at any reference that has the same value as any of FragmentSubstitution.replacement's ReplacementBoundaryElement that are of type FromReplacement (using the model element references called outsideBoundaryElement).If multi is true, then a number of copies of the replacement fragment will be copied onto the placement. The resolution model will define how many.Any substitutions addressing placements inside the given replacement fragment will be performed on the copy of the replacement fragment which is the last one generated.3. Binding boundary elements. The placement and replacement boundary elements are connected by bindings. The bindings are given by the BoundaryElementBindings:
1. FromBinding: fromReplacement.insideBoundaryElement.propertyName[] = fromPlacement.outsideBoundaryElement[]
2. ToBinding: toPlacement.outsideBoundaryElement.propertyName[] = toReplacement.insideBoundaryElement[]. This definition in fact also covers attributes that have multiplicity. Such attributes may be seen as arrays or collections, and repeated reference assignments to such attributes during variability transformation will mean adding a new individual reference to the identifier collection.

#### 11.2.16.2 Generalizations

ChoiceVariationPoint

RepeatableVariationPoint

#### 11.2.16.3 Attributes

- multi : Boolean [1..1]
  Indicates multiple fragment substitution.

#### 11.2.16.4 Associations

- boundaryElementBinding : BoundaryElementBinding [0..*]
  Specifies the binding between the placement and replacement fragments.
- placement : PlacementFragment [1..1]
  Specifies the fragment to be replaced.
- replacement : ReplacementFragmentType [0..1]
  Specifies the replacement to be utilized.

#### 11.2.16.5 Semantics

Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a Fragment Substitution, the placement fragment must exist in the base model.
Post-condition:
After a FragmentSubstitution has been executed, the placement elements cannot be found in the resolved model whereas the replacement elements are in the resolved model.
OCL :
context FragmentSubstitution::eval(ctx : CVLExecutionContext)
pre :

self.boundaryElementBinding->forAll(boundaryElt |boundaryElt.oclIsTypeOf(ToBinding) implies boundaryElt.oclAsType(ToBinding).toPlacement.insideBoundaryElement->forAll(e | ctx.resolvedModelElements->includes(e)) )
post :
self.boundaryElementBinding->forAll(boundaryElt |boundaryElt.oclIsTypeOf(ToBinding) implies boundaryElt.oclAsType(ToBinding).toPlacement.insideBoundaryElement->forAll(e | not ctx.resolvedModelElements->includes(e)) )
and self.boundaryElementBinding->forAll(boundaryElt |boundaryElt.oclIsTypeOf(ToBinding) implies boundaryElt.oclAsType(ToBinding).toReplacement.insideBoundaryElement->forAll(e | ctx.resolvedModelElements->includes(e)) )
Note : FragmentSubstitution needs the use of FromBinding and ToBinding metaclasses (please refer to the corresponding pre and post conditions).

## 11.2.17 FromBinding

### 11.2.17.1 Description

FromBinding defines a binding between boundary elements of kind FromPlacement/FromReplacement. The FromPlacement boundary
element that has to be bound to the FromPlacement.

### 11.2.17.2 Generalizations

BoundaryElementBinding

### 11.2.17.3 Associations

- fromPlacement : FromPlacement [1..1]
  Specifies the FromPlacement boundary element that is part of the binding.
- fromReplacement : FromReplacement [1..1]
  Specifies the FromReplacement boundary element that is part of the binding.

### 11.2.17.4 Semantics

Invariant
Invariant : The FromPlacement boundary element must be bound to the FromReplacement element.
OCL :
-- The FromPlacement boundary element must be bound to the FromReplacement element
context FromBinding :
inv mustBeBoundToTheReplacement :
self.fromPlacement.fromReplacement = self.fromReplacement
Dynamic semantics
Pre and post condition
Pre-condition :
None
Post-condition:
After a FragmentSubstitution has been executed (and as a consequence a FromBinding), the elements inside the replacement element references the element outside the placement elements. These outside elements referenced before the placement elements.
OCL :
-- FromBinding
-- (FragmentSubstitution : the placement and replacement boundary elements are connected by bindings)
-- fromReplacement.insideBoundaryElement.propertyName[] = fromPlacement.outsideBoundaryElement[]
context FromBinding::eval(ctx : CVLExecutionContext)

pre :
post :
self.fromReplacement.insideBoundaryElement.getPropertyValue(self.fromReplacement.propertyName)->forAll( val | val.oclIsTypeOf(ObjectHandle) and self.fromPlacement.outsideBoundaryElement->includes(val.oclAsType(ObjectHandle)))

## 11.2.18 FromPlacement

### 11.2.18.1 Description

FromPlacement is the kind of boundary element that defines the outwards boundary of the owning placement fragment. The outsideBoundaryElement refers to the model elements on the outside of the placement fragment. In a fragment substitution these have to be referred by model elements within the replacement fragment.

### 11.2.18.2 Generalizations

PlacementBoundaryElement

### 11.2.18.3 Associations

- fromReplacement : FromReplacement [0..1]
  Reference to a FromReplacement in a containing replacement fragment.
- outsideBoundaryElement : ObjectHandle [0..*]
  Outside Model Element that are referred by the model elements inside the placement fragment.

## 11.2.19 FromReplacement

### 11.2.19.1 Description

FromReplacement is the kind of boundary element that defines the outwards boundary of the owning replacement fragment. propertyName is the name of the reference attribute of inside boundary model element that will be changed as part of a fragment substitution. The insideBoundaryElements refer to the base model elements that will have their reference attributes updated as part of a fragment substitution. The outsideBoundaryElement refers to the model elements on the outside of the replacement fragment. In a fragment substitution these references are used to define the extent of the replacement fragment.

### 11.2.19.2 Generalizations

ReplacementBoundaryElement

### 11.2.19.3 Attributes

- propertyName : String [1..1]
  Name of the attribute to be changed.

### 11.2.19.4 Associations

- fromPlacement : FromPlacement [0..1]
  Reference to a FromPlacement contained by the replacement fragment.

- insideBoundaryElement : ObjectHandle [0..1]
  Inside model elements that refer outside model elements.
- outsideBoundaryElement : ObjectHandle [0..*]
  Outside model elements that are referred by model elements inside the fragment. Used to distingui sh multiplicity references.

## 11.2.20 IntegerLiteralExp

### 11.2.20.1 Description

A literal expression that represents integer numbers.

### 11.2.20.2 Generalizations

NumericLiteralExp

### 11.2.20.3 Attributes

- integer : Integer [1..1]
  The integer value of the IntegerLiteralExpression.

## 11.2.21 LinkEndSubstitution

### 11.2.21.1 Description

A LinkEndSubstitution is a choice variation point which specifies that an object in the base model, called the replacement, may be substituted for another as the end of a particular link in the base model. The link-end is identified via a link handle referencing the base model plus a string representing a MOF property owned by the association of which the link is an instance. The placement is specified via an object handle.When this variation point is applied,the link-end is redirected to the replacement object.

### 11.2.21.2 Generalizations

ChoiceVariationPoint

### 11.2.21.3 Attributes

- linkEndIdentifier : String [1..1]
  The name of the MOF Property in the object's metaclass identifying the link-end where this substitution occurs.

### 11.2.21.4 Associations

- link : LinkHandle [1..1]
  Reference to the link.
- replacementObject : ObjectHandle [1..1]
  The object handle identifying the base model object replacing the link-end in this substitution. If no replacement is specified then the object is expected to arrive in a variable, that is the variation point must be bound to a variable to be acted on.

#### 11.2.21.5 Semantics

Dynamic semantics
Pre and post condition
Pre-condition :
None
Post-condition:
After a Link Assignment has been executed, the newEnd object handle must be an end of the LinkHandle link.
Note : To express more easily this OCL pre condition, we add an operation getEnd() on the LinkHandle metaclass. This operation computes the respective end of this linkHandle.
OCL :

```
-- LinkAssignment
context LinkAssignment::eval(ctx : CVLExecutionContext)
pre :
post : self.link.getEnd()->includes(self.newEnd)
```

### 11.2.22 LinkExistence

#### 11.2.22.1 Description

LinkExistence is a choice variation point which indicates that a link of the base model may or may not exist in the materialized model. The link is identified via a link handle. When this variation point is applied the link identified in the base model as the optionalLink t is deleted.

#### 11.2.22.2 Generalizations

ChoiceVariationPoint

#### 11.2.22.3 Associations

- optionalLink : LinkHandle [1..1]
  The link handle identifying the base model link whose existence is in question

#### 11.2.22.4 Semantics

Invariant : A LinkExistence must be bound to a Choice, not to a variable or to a classifier
OCL :

```
-- LinkExistence
-- A LinkExistence must be bound to a Choice, not to a variable or to a classifier
context LinkExistence :
inv linkExistence_must_bound_to_choice :
(self.bindingVspec->exists( vspec | vspec.oclIsTypeOf(Choice)))
and (not (self.bindingVspec->exists( vspec | vspec.oclIsTypeOf(Variable))))
and (not (self.bindingVspec->exists( vspec | vspec.oclIsTypeOf(VClassifier))))
```

Dynamic semantics
Pre and post condition
Pre-condition :
None
Post-condition:
After a LinkExistence has been executed, the optionalLink must be in the resolved model otherwise it cannot be in the resolved model.
OCL :

```
-- LinkExistence
context LinkExistence::eval(ctx : CVLExecutionContext)
pre :
```

post:
ctx.selectedVSpecs->includes((self.bindingVspec->asOrderedSet()->first())) implies ctx.baseLinks->includes(self.optionalLink) and
ctx.unselectedVSpecs->includes(self.bindingVspec->asOrderedSet()->first()) implies not ctx.baseLinks->includes(self.optionalLink)

## 11.2.23 LinkHandle

### 11.2.23.1 Description

A link handle identifies a link of the base model. This Class abstracts over the cross-domain referencing mechanism needed to refer from CVL elements to base model links.

### 11.2.23.2 Generalizations

BaseModelHandle

### 11.2.23.3 Attributes

- MOFRef : String [1..1]
  Representing a MOF Reference.

## 11.2.24 MultiplicityInterval

### 11.2.24.1 Description

A MultiplicityInterval specifies lower and upper multiplicities.

### 11.2.24.2 Attributes

- lower : Integer [1..1]
  The lower multiplicity.
- upper : Integer [1..1]
  The upper multiplicity.

### 11.2.24.3 Semantics

Invariant :
The value of the lower multiplicity must be inferior or equal to the upper multiplicity
OCL :
-- MultiplicityInterval
-- lower_inferior_upper : The value of the lower multiplicity must be inferior or equal to the upper multiplicity
context MultiplicityInterval :
inv lower_inferior_upper :
(self.upper == (-1))
or (self.lower <> -1 and self.upper <> -1 and self.lower <= self.upper)

## 11.2.25 NamedElement

### 11.2.25.1 Description

An named element is an element identifiable by name. Names are composed of letters, numbers, the underscore sign "_" and the dollar sign "$". The first character of a name must be a letter, an underscore or a dollar sign. Reserved keywords of the constraint language cannot be used as identifiers.

### 11.2.25.2 Attributes

- name : String [1..1]
  The name of the element. Names are composed of letters, numbers, the underscore sign "_" and the dollar sign "$". The first character of a name must be a letter, an underscore or a dollar sign. Reserved keywords of the constraint language cannot be used as identifiers.

## 11.2.26 NumericLiteralExp

### 11.2.26.1 Description

A literal expression that represents real, unlimited natural, and integer constants.

### 11.2.26.2 Generalizations

BCLExpression

## 11.2.27 ObjectExistence

### 11.2.27.1 Description

ObjectExistence is a choice variation point which indicates that an object of the base model may or may not exist in the materialized model. The object is identified via an object handle. When this variation point is applied the object identified in the base model as the optionalObject is deleted.

### 11.2.27.2 Generalizations

ChoiceVariationPoint

### 11.2.27.3 Associations

- optionalObject : ObjectHandle [1..1]
  The object handle identifying the base model object whose existence is in question.

### 11.2.27.4 Semantics

Invariant
Invariant : An ObjectExistence must be bound to a Choice, not to a variable or to a classifier
OCL :

```
-- ObjectExistence
-- An ObjectExistence must be bound to a Choice, not to a variable or to a classifier
context ObjectExistence :
inv must_bound_to_choice :
(self.bindingVspec->exists( vspec | vspec.oclIsTypeOf(Choice)))
and not ( (self.bindingVspec->exists( vspec | vspec.oclIsTypeOf(Variable))))
and not ( (self.bindingVspec->exists( vspec | vspec.oclIsTypeOf(VClassifier))))
```

Dynamic semantics
Pre and post condition
Pre-condition :
None
Post-condition:
After an ObjectExistence has been executed, the optionalObject must be in the resolved model otherwise it cannot be in the resolved model.
OCL :
-- ObjectExistence
context ObjectExistence::eval(ctx : CVLExecutionContext)
pre :
post:
ctx.selectedVSpecs->includes(self.bindingVspec->asOrderedSet()->first()) implies
ctx.resolvedModelElements->includes(self.optionalObject) and
ctx.unselectedVSpecs->includes((self.bindingVspec->asOrderedSet()->first())) implies not
ctx.resolvedModelElements->includes(self.optionalObject)

## 11.2.28 ObjectHandle

### 11.2.28.1 Description

An object handle identifies an object of the base model. This Class abstracts over the cross-domain referencing mechanism needed to refer from CVL elements to base model objects.

### 11.2.28.2 Generalizations

BaseModelHandle

### 11.2.28.3 Attributes

- MOFRef : String [1..1]
  Representing a MOF Reference.

## 11.2.29 ObjectSpecification

### 11.2.29.1 Description

An ObjectSpecification specifies a value which is an object of the base mode through an object handle.

### 11.2.29.2 Generalizations

ValueSpecification

### 11.2.29.3 Associations

- object : ObjectHandle [1..1]
  The object specified.
- type : ObjectType [1..1]
  Type of the object.

## 11.2.30 ObjectSubstitution

### 11.2.30.1 Description

An ObjectSubstitution is a choice variation point which specifies that an object of the base model, called the replacement, may be substituted for another, called the placement. The placement and replacement objects are specified via object handles identifying base model objects.When this variation point is applied, all links touching the placement are redirected to the replacement and the placement is deleted.

### 11.2.30.2 Generalizations

ChoiceVariationPoint

### 11.2.30.3 Associations

- placementObject : ObjectHandle [1..1]
  The object handle identifying the base model object to be replaced by the replacement object in this substitution.
- replacementObject : ObjectHandle [1..1]
  The object handle identifying the base model object replacing to the placement object in this substitution. If no replacement is specified then the object is expected to arrive in a variable, that is the variation point must be bound to a variable to be acted on.

### 11.2.30.4 Semantics

Invariant : An ObjectSubstitution may not be bound to a VClassifier
OCL :
-- ObjectSubstitution
-- An ObjectSubstitution may not be bound to a VClassifier
context ObjectSubstitution :
inv notBoundToAVClassifier :
not (self.bindingVspec->exists( vspec | vspec.oclIsTypeOf(VClassifier)))
Dynamic semantics
Pre and post conditions
Pre-condition :
Before the execution of an ObjectSubstitution, placement and replacement object must exist in the base model.
Post-condition:
After an Object Substitution has been executed, the replacement object must exist in the resolved model whereas the placement object cannot be found in the resolved model.
OCL :
-- ObjectSubstitution
context ObjectSubstitution::eval(ctx : CVLExecutionContext)
pre:
ctx.resolvedModelElements->includes( self.placementObject)
and ctx.resolvedModelElements->includes( self.replacementObject)
post:
ctx.resolvedModelElements->includes( self.replacementObject)
and not ctx.resolvedModelElements->includes( self.placementObject)

## 11.2.31 ObjectType

### 11.2.31.1 Description

A type of objects in the base model, specified as a metaclass in the metamodel of which the base model in an instance.

### 11.2.31.2 Generalizations

Variabletype

### 11.2.31.3 Attributes

- metaClass : String [1..1]
  The name of the metaclass in the metamodel of which the base model is an instance.

## 11.2.32 OpaqueConstraint

### 11.2.32.1 Description

A Constraint imposes additional restrictions that cannot be expressed in the base language. Each constraint has associated a VSpec context. It allows expressing universal quantification without explicit quantification phrases such as "for all elements belonging to...'. Constraints written in a context are applied to each named VSpec that is available in this context.The full constraint language (with classifiers) has set semantics for a ll non-
propositional expressions (VSpecs, Integers, etc.). The small language (without classifiers) relies mostly on propositional formulas. A Constraint contains GeneralExpressions, which are always expected to evaluate t o a boolean value. In case of set semantics GeneralExpressions are always LogicalExps, while in the small l anguage they are LogicalExps or VSpecReferences. In the small language VSpecReferences have propositi onal semantics and are treated as Boolean choices.

### 11.2.32.2 Generalizations

Constraint

### 11.2.32.3 Attributes

- constraint : String [1..1]
  Constraint as an opaque String.
- constraintLanguage : String [1..1]
  Language of the OpaqueConstraint as a String.

### 11.2.32.4 Semantics

Invariant :
Each constraint must have a VSpec context
OCL :

```
-- Constraint
-- Each Constraint must have a VSpec context
context Constraint :
inv hasAVSpecContext :
not (self._context == (null))
```

## 11.2.33 OpaqueVariationPoint

### 11.2.33.1 Description

An OpaqueVariationPoint is an executable, domain-specific variation point whose semantics is not defined by CVL. It is the responsibility of the specific domain to execute this kind of variation point.
If bound to a choice then an OpaqueVariation point will be executed upon a positive decision. If bound to a VClassifier then it will be executed once for each instance created from it. If bound to a variable then it will be executed when a value is assigned to it, also providing the value as parameter for the execution.

### 11.2.33.2 Generalizations

VariationPoint

VariationPoint

### 11.2.33.3 Associations

- placeHolder : BaseModelHandle [1..1]
  The place holder of the OpaqueVariationPoint.
- sourceObject : ObjectHandle [1..*]
  The source objects of the OpaqueVariationPoint.
- type : OVPType [1..1]
  The transformation used by the opaque variation point.

### 11.2.33.4 Semantics

Dynamic semantics
The OpaqueVariationPoint allows user to make use of variations that are not directly defined by CVL. So, its semantics is not detailed here, and we do not have pre and post conditions
Pre and post condition
Pre-condition :
None
Post-condition:
None
OCL :
-- OpaqueVariationPoint
context OpaqueVariationPoint::eval(ctx : CVLExecutionContext)
pre : true
post : true

## 11.2.34 <Enumeration>Operation

### 11.2.34.1 Description

Enumerates operations available in basic constraint language: logNot (logical negation), logAnd (logical conjunction), logOr (logical disjunction), logImplies (logical implication), logXor (logical exclusive-or), arithPlus (arithmetic addition), arithMinus (arithmetic subtraction), arithNeg (arithmetic negation), arithMult (arithmetic multiplication), arithDiv (arithmetic division), strConc (string concatenation), eq (equality), lte (less than or equal), gte (greater than or equal), lt (less than), gt (greater than), isDefined (checks if value is not bottom), isUndefined (checks if value is bottom)

#### 11.2.34.2 Literals

- arithDev _Ref304197604
  Arithmetic division
- arithMinus _Ref304197604
  Arithmetic subtraction
- arithMult _Ref304197604
  Arithmetic multiplication
- arithNeg _Ref304197604
  Arithmetic negation
- arithPlus _Ref304197604
  Arithmetic addition
- eq _Ref304197604
  Equal
- gt _Ref304197604
  Greater than
- gte _Ref304197604
  Greater than or equal
- isDefined _Ref304197604
  Checks if the value is defined.
- isUndefined _Ref304197604
  Check if the value is undefined.
- logAnd _Ref304197604
  Logical and
- logIff _Ref304197604
  Logical if and only if
- logImplies _Ref304197604
  Logical implies
- logNot _Ref304197604
  Logical not
- logOr _Ref304197604
  Logical or
- logXor _Ref304197604
  Logical xor
- lt _Ref304197604
  Logical less than

- lte _Ref304197604
  Logical less than or equal
- strConc _Ref304197604
  String concatenation

## 11.2.35 OperationCallExp

### 11.2.35.1 Description

An expression that represents operations on given subexpressions (arguments). Operations include logical, arithmetic, relational operations, and two predicates.

### 11.2.35.2 Generalizations

BCLExpression

### 11.2.35.3 Associations

- argument : BCLExpression [0..*]
  Expressions for the arguments of the operation to be called.
- operation : Operation [1..1]
  Operation to be called.

## 11.2.36 OVPSemanticSpec

### 11.2.36.1 Description

OVPSermanticSpec describes a transformation which will be performed by an OpaqueVariationPoint. The transformation is given as a string containing a textual definition of the transformation in the given transformation language. The transformation language is also given in the OVPSemanticSpec.

### 11.2.36.2 Attributes

- modelTransformation : String [1..1]
  Model Transformation specification as String.
- transfromationLanguage : String [1..1]
  Language of the model transformation.

## 11.2.37 OVPType

### 11.2.37.1 Description

OVPType (Opaque Variation Point type) is a model transformation pattern which may be used to define an opaque variation point.

### 11.2.37.2 Generalizations

VPackageable

#### 11.2.37.3 Associations

- spec : OVPSemanticSpec [1..1]
  Reference to the specification.

### 11.2.38 ParametricLinkEndSubstitution

#### 11.2.38.1 Description

A ParametricLinkEndSubstitution is a parametric variation point which specifies that an object arriving as parameter, called the replacement, will be substituted for an object of the base model as the end of a particular link in the base model. The link-end is identified via a link handle pointing to the base model plus a string representing a MOF property owned by the association of which the link is an instance. The variation point must be bound to a variable which will provide the replacement object coming as parameter. When this variation point is applied,the link-end is redirected to the replacement coming as parameter.

#### 11.2.38.2 Generalizations

ParametricVariationPoint

#### 11.2.38.3 Attributes

- linkEndIdentifier : String [1..1]
  Name of the link end.

#### 11.2.38.4 Associations

- link : LinkHandle [1..1]
  Reference to the link end.

#### 11.2.38.5 Semantics

Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a ParametricLinkAssignment, the link link must exist in the base model and a end corresponding to the given linkEndIdentifier exists in the ObjectHandle.
Post-condition:
After a ParametricLinkAssignment has been executed, the object which arrives in the variable is at the end of the link link.
OCL :

```
-- ParametricLinkAssignment
-- A end corresponding to the given linkEndIdentifier exists in the ObjectHandle
context ParametricLinkAssignment::eval(ctx : CVLExecutionContext)
pre : ctx.baseLinks->includes(self.link)
and if self.getPropertyValue(self.linkEndIdentifier)
null
then
self.getPropertyValue(self.linkEndIdentifier)->forAll( val | ctx.resolvedModelElements-
>includes(val.oclAsType(ObjectHandle)))
else
false
endif
post: self.link.getEnd()->includes(
```

VariableValueAssignment.allInstances()->select (varValueAssign | (self.bindingVspec
->includes(varValueAssign.resolvedVariable)))->asOrderedSet()->first().value )

## 11.2.39 ParametricObjectSubstitution

### 11.2.39.1 Description

A ParametricObjectSubstitution is a parametric variation point which specifies that an object arriving as parameter, called the replacement, will be substituted for an object of the base model, called the placement. The placement object is identified via an object handle identifying a base model object. The variation point must be bound to a variable which will provide the replacement coming as parameter.When this variation point is applied, all links touching the placement are redirected to the replacement and the placement is deleted.
An ObjectSubstitution may not be bound to a choice or VClassifier.

### 11.2.39.2 Generalizations

ParametricVariationPoint

### 11.2.39.3 Associations

- placementObject : ObjectHandle [1..1]
  The placement object.

### 11.2.39.4 Semantics

Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a ParametricObjectSubstitution, the placement object must exist in the base model and the replacement object is in the variable.
Post-condition:
After a ParametricObjectSubstution has been executed, the replacement object must exist in the resolved model whereas the placement object cannot be found in the resolved model.
OCL :
-- ParametricObjectSubstitution
context ParametricObjectSubstitution::eval(ctx : CVLExecutionContext)
pre:
ctx.resolvedModelElements->includes( self.placementObject)
and (VariableValueAssignment.allInstances()->select (varValueAssign | (self.bindingVspec
->includes(varValueAssign.resolvedVariable)))->asOrderedSet()->first().value)
null
post:
ctx.resolvedModelElements->includes( (VariableValueAssignment.allInstances()->select (varValueAssign | (self.bindingVspec
->includes(varValueAssign.resolvedVariable)))->asOrderedSet()-
>first().value).oclAsType(ObjectSpecification).object)
and not ctx.resolvedModelElements->includes( self.placementObject)

## 11.2.40 ParametricSlotAssignmet

### 11.2.40.1 Description

A parametric slot assignment is a parametric variation point which specifies that a value arriving as parameter will be assigned to a particular slot in a particular object in the base model. The object is

identified via an object handle pointing to the base model, The object is identified via an object handle, and the the slot is identified via its name, as indicated in the attribute slotIdentifie. The variation point must be bound to a variable which will provide the value coming as parameter.
When this variation point is applied, the value coming as parameter is inserted into the base model slot

#### 11.2.40.2 Generalizations

ParametricVariationPoint

#### 11.2.40.3 Attributes

- slotIdentifier : String [1..1]
  The name of the slot identifier.

#### 11.2.40.4 Associations

- slotOwner : ObjectHandle [1..1]
  The slot owner.

#### 11.2.40.5 Semantics

Invariant
Invariant : The slotIdentifier must correspond to a property name of the associated ObjectHandle
OCL :
-- ParametricSlotAssignment
-- The slotIdentifier must correspond to a property name of the associated ObjectHandle
context ParametricSlotAssignmet :
inv slotIdentifierExists :
self.slotOwner.getPropertyByName(self.slotIdentifier)
null
Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a ParametricSlotAssignment, the ObjectHandle slotOwner must exist in the base model.
Post-condition:
After a ParametricSlotAssignment has been executed, the slot identified by the slotIdentifier and contained in the ObjectHandle slotOwner is assigned with a given value.
OCL :
-- ParametricSlotAssignment
context ParametricSlotAssignmet::eval(ctx : CVLExecutionContext)
pre : ctx.resolvedModelElements->includes(self.slotOwner)
post: self.slotOwner.getPropertyValue(self.slotIdentifier)->asOrderedSet()->first() ==
(VariableValueAssignment.allInstances()->select (varValueAssign | (self.bindingVspec
->includes(varValueAssign.resolvedVariable)))->asOrderedSet()->first().value)

### 11.2.41 ParametricVariationPoint

#### 11.2.41.1 Description

A parametric variation point is a variation point that depends on a parameter and must be bound to a variable. During materialization the value supplied as the resolution for the variable is used for the parameter.

#### 11.2.41.2 Generalizations

VariationPoint

#### 11.2.41.3 Associations

- bindingVariable : Variable [1..1]
  The binding variable.

### 11.2.42 PlacementBoundaryElement

#### 11.2.42.1 Description

Represents the boundary between a placement fragment and the rest of the base model.

### 11.2.43 PlacementFragment

#### 11.2.43.1 Description

A PlacementFragment defines a fragment (set of model elements) of the base model that will be replaced by a ReplacementFragment during the variability transformation. The set of model elements of the fragment will be deleted.

#### 11.2.43.2 Generalizations

VariationPoint

#### 11.2.43.3 Associations

- placementBoundaryElement : PlacementBoundaryElement [0..*]
  The boundary elements captures all the relations from and to the fragment.

### 11.2.44 <Enumeration>PrimitiveTypeEnum

#### 11.2.44.1 Description

An enumeration of the most primitive types: String, Boolean, Integer, Real and UnlimitedNatural.

#### 11.2.44.2 Literals

- Boolean _Ref304197604
  Primitive type Boolean literal.
- Integer _Ref304197604
  Primitive type Integer literal.
- Real _Ref304197604
  Primitive type Real literal.
- String _Ref304197604
  Primitive type String literal.

- UnlimitedNatural _Ref304197604
  Primitive type UnlimitedNatural literal.

## 11.2.45 PrimitiveValueSpecification

### 11.2.45.1 Description

A PrimitiveValueSpecification contains an expression in our Basic Constraint Language and is typed by a primitive type.

### 11.2.45.2 Generalizations

ValueSpecification

### 11.2.45.3 Associations

- expression : BCLExpression [1..1]
  Expression specifying the value.
- type : PrimitveType [0..1]
  The PrimitiveType of this expression.

## 11.2.46 PrimitveType

### 11.2.46.1 Description

A type of a variable which is either String, Integer,UnlimitedNatural, Real, or Boolean.

### 11.2.46.2 Generalizations

Variabletype

### 11.2.46.3 Attributes

- type : _kt23ZuUlEeGkDLEFB5Suyg [1..1]
  The primitive type as an enumeration value.

## 11.2.47 RealLiteralExp

### 11.2.47.1 Description

A literal expression that represents floating-point numbers.

### 11.2.47.2 Generalizations

NumericLiteralExp

### 11.2.47.3 Attributes

- real : TODO ERROR [1..1]
  The real value of this RealLiteralExpression.

## 11.2.48 RepeatableVariationPoint

### 11.2.48.1 Description

A repeatable variation point is a variation point that may be applied several times during materialization. It may be bound to a VClassifier and is applied once for each instance of it.

### 11.2.48.2 Generalizations

VariationPoint

### 11.2.48.3 Associations

- bindingClassifier : VClassifier [0..1]
  The binding classifier.

## 11.2.49 ReplacementBoundaryElement

### 11.2.49.1 Description

Represents the boundary between a replacement fragment and the rest of the base model.

## 11.2.50 ReplacementFragmentSpecification

### 11.2.50.1 Description

A value of ReplacementFragmentType

### 11.2.50.2 Generalizations

ValueSpecification

### 11.2.50.3 Associations

- type : ReplacementFragmentType [1..1]
  The corresponding ReplacementFragmentType

## 11.2.51 ReplacementFragmentType

### 11.2.51.1 Description

Replacement Fragment Type defines a fragment of the base model that will be used as replacement for som e placement fragment
of the base model.Constraints:The placements contained in a replacement fragments should only involve model elements which are inside the replacement fragment. These placements can be used in all instances of a replacement fragment.Semantics:The semantics of Replacement Fragment Type can be found under Fragment Substitution.

### 11.2.51.2 Generalizations

Variabletype

#### 11.2.51.3 Associations

- placementFragment : PlacementFragment [0..*]
  Set of placements contained by the replacement fragment.
- replacementBoundaryElement : ReplacementBoundaryElement [0..*]
  The boundary elements captures all the relations from and to the fragment.

#### 11.2.51.4 Semantics

Invariant :
OCL :

### 11.2.52 SlotAssignment

#### 11.2.52.1 Description

A slot assignment is a choice variation point which specifies a value to be assigned to a particular slot in a particular object in the base model. The object is identified via an object handle pointing to the base model, and the slot is identified via its name, stored in the slotIdentifier attribute. The value to be assigned is specified explicitly.When this variation point is applied, the specified value is inserted into the base model slot.

#### 11.2.52.2 Generalizations

ChoiceVariationPoint

#### 11.2.52.3 Attributes

- slotIdentifier : String [1..1]
  The name of the MOF Property in the object's metaclass identifying the slot to which the value is to be assigned.

#### 11.2.52.4 Associations

- slotOwner : ObjectHandle [1..1]
  The object handle identifying the base model object to whose slot the value is to be assigned.
- value : ValueSpecification [0..1]
  The value to be assigned.

#### 11.2.52.5 Semantics

Invariant
Invariant : The property named as self.slotIdentifier must exist in the slotOwner object
OCL :

```
-- SlotAssigmnent
-- The property named as self.slotIdentifier must exist in the slotOwner object
context SlotAssignment :
inv propertyIn_slotOwner :
if self.SlotOwner
null
then
self.SlotOwner.getPropertyByName(self.slotIdentifier)
```

null
else
-- if SlotOwner does not exists neither the property
false
endif
Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a SlotAssignment, the SlotOwner object must exist in the base model and it must have a MOF property called as in the slotIdentifier.
Post-condition:
After a SlotAssignment has been executed, the MOF property called as in the slotIdentifier in the SlotOwner has been assigned with the value value.
Note : To express more easily this OCL pre condition, we add the operations getPropertyByName(String propertyName) and getPropertyValue(String propertyName) on the ObjectHandle metaclass to obtain respectively the property with the given name and the value associated with this property.
OCL :
context SlotAssignment::eval(ctx : CVLExecutionContext)
pre : self.SlotOwner
null and ctx.resolvedModelElements->includes(self.SlotOwner) and self.SlotOwner->asOrderedSet()->first().getPropertyByName(self.slotIdentifier)
null
post : self.SlotOwner->asOrderedSet()->first().getPropertyValue(self.slotIdentifier)->asOrderedSet()->first() == (self.value)

## 11.2.53 SlotValueExistence

### 11.2.53.1 Description

SlotValueExistence is a choice variation point which indicates that a value in a slot of some object of the base model may or may not exist in the materialized model. The object is identified via an object handle and the the slot is identified via its name, as indicated in the attribute slotIdentifie.When this variation point is applied, the slot in the base model is cleared from whatever value it has.

### 11.2.53.2 Generalizations

ChoiceVariationPoint

### 11.2.53.3 Attributes

- slotIdentifier : String [1..1]
  The name of the slot identifier.

### 11.2.53.4 Associations

- slotOwner : ObjectHandle [1..1]
  Reference to the slot owner.

### 11.2.53.5 Semantics

Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a SlotValueExistence, the slotOwner element must exist in the base model.
Post-condition:

<blockquote>After a SlotValueExsitence has been executed, the MoF property with the name such as in the slotIdentifier is in the ObjectHandle otherwise it cannot be found in the ObjectHandle.
</blockquote>OCL :
-- SlotValueExistence
context SlotValueExistence::eval(ctx : CVLExecutionContext)
pre : ctx.resolvedModelElements->includes(self.slotOwner)
post :
ctx.selectedVSpecs->includes(self.bindingVspec->asOrderedSet()->first()) implies self.slotOwner->asOrderedSet()->first().getPropertyByName(self.slotIdentifier)
null
and
ctx.unselectedVSpecs->includes(self.bindingVspec->asOrderedSet()->first()) implies (self.slotOwner->asOrderedSet()->first().getPropertyByName(self.slotIdentifier)) == (null)

## 11.2.54 StringLiteralExp

### 11.2.54.1 Description

A literal expression that represents strings.

### 11.2.54.2 Generalizations

BCLExpression

### 11.2.54.3 Attributes

- string : String [1..1]
  The string value of this StringLiteralExpression.

## 11.2.55 ToBinding

### 11.2.55.1 Description

ToBinding defines a binding between boundary elements of kind ToPlacement/ToReplacement. The ToPlacement boundary element
has to be bound to the ToReplacement.

### 11.2.55.2 Generalizations

BoundaryElementBinding

### 11.2.55.3 Associations

- toPlacement : ToPlacement [1..1]
  Specifies the ToPlacement boundary element that is part of the binding.
- toReplacement : ToReplacement [1..1]
  Specifies the ToReplacement boundary element that is part of the binding.

### 11.2.55.4 Semantics

Invariant
Invariant : The ToPlacement boundary element must be bound to the ToReplacement element
OCL :

-- The ToPlacement boundary element must be bound to the ToReplacement element
context ToBinding :
inv mustBeBoundToTheToReplacement :
self.toPlacement.toReplacement = self.toReplacement
Dynamic semantics
Pre and post condition
Pre-condition :
None
Post-condition:
After a FragmentSubstitution has been executed (and as a consequence a ToBinding), the elements that referenced before the placement inside model elements reference now the replacement inside model elements.
OCL :
-- ToBinding
-- (FragmentSubstitution : the placement and replacement boundary elements are connected by bindings)
-- toPlacement.outsideBoundaryElement.propertyName[] = toReplacement.insideBoundaryElement[]
context ToBinding::eval(ctx : CVLExecutionContext)
pre :
post : self.toPlacement.outsideBoundaryElement.getPropertyValue(self.toPlacement.propertyName)->forAll( val | val.oclIsTypeOf(ObjectHandle) and self.toReplacement.insideBoundaryElement->includes(val.oclAsType(ObjectHandle)))

## 11.2.56 ToPlacement

### 11.2.56.1 Description

ToPlacement is the kind of boundary element that defines the boundary between the owning placement fragment and the rest of the base model. The insideBoundaryElements denote the ModelElements of owning fragment that are referred to by outside model elements. The outsideBoundaryElement together with the propertyName denotes the attributes of model elements on the outside of the placement fragment that refer to the inside boundary model elements.Constraints:insideBoundaryElement = outsideRef.insideBoundaryElement outsideBoundaryElement != null xor outsideRef != null

### 11.2.56.2 Generalizations

PlacementBoundaryElement

### 11.2.56.3 Attributes

- propertyName : String [1..1]
  Name of the attribute to be changed.

### 11.2.56.4 Associations

- insideBoundaryElement : ObjectHandle [0..*]
  Model elements that are referred to by outside model elements. Used to distinguish miltiplicity references.
- outsideBoundaryElement : ObjectHandle [0..1]
  Outside model elements that refer model elements inside the fragment.
- toReplacement : ToReplacement [0..1]
  Reference to a ToReplacement in a containing replacement fragment.

#### 11.2.56.5 Semantics

Invariant :
<p > Constraint :
<p >1) self.outsideBoudaryElement <> null xor outsideRef != null
<p >2) All outsideBoundaryElement point on insideBoundaryElement
OCL :
-- ToPlacement
-- The outsideBoundaryElement together with the propertyName denotes the attributes of model elements outside of the placement fragment that refer to the inside boundary model elements
-- Constraint :
-- 1) self.outsideBoudaryElement <> null xor outsideRef != null
-- 2) All outsideBoundaryElement point on insideBoundaryElement
context ToPlacement :
inv insideBoundaryElements_outsideRef :
-- 1) self.outsideBoudaryElement <> null
(not self.outsideBoundaryElement->isEmpty()
xor self.outsideBoundaryElement.getPropertyValue(self.propertyName) <> null)
and
-- 2) All outsideBoundaryElement point on insideBoundaryElement
self.outsideBoundaryElement.getPropertyValue(self.propertyName)->forAll( val |
self.insideBoundaryElement->includes(val.oclAsType(ObjectHandle) ))

### 11.2.57 ToReplacement

#### 11.2.57.1 Description

ToReplacement is the kind of boundary element that defines the inwards boundary of the owning replacement fragment. The insideBoundaryElement defines the starting points for the traversal to isolate the model elements that as part of a fragment substitution will be copied into the placement fragment.

#### 11.2.57.2 Generalizations

ReplacementBoundaryElement

#### 11.2.57.3 Associations

- insideBoundaryElement : ObjectHandle [0..*]
  Model elements that are referred to by outside model elements.
- toPlacement : ToPlacement [0..1]
  Reference to a ToPlacement contained by the replacement fragment.

### 11.2.58 UnlimitedLiteralExp

#### 11.2.58.1 Description

A literal expression that represents unlimited natural numbers.

#### 11.2.58.2 Generalizations

NumericLiteralExp

#### 11.2.58.3 Attributes

- unlimited : UnlimitedNatural [1..1]
  Value of this UnlimitedLiteralExpression.

### 11.2.59 ValueSpecification

#### 11.2.59.1 Description

A ValueSpecification specifies a value which is either primitive, or an object of the base mode, or a fragment of the base model.

#### 11.2.59.2 Associations

- type : Variabletype [1..1]
  The type of the ValueSpecification.

### 11.2.60 Variable

#### 11.2.60.1 Description

A variable is a VSpec whose resolution requires providing a value of its specified type. When a parametric variation point is bound to a variable, the value provided for the variable as resolution will be used as the actual parameter when applying the variation point during materialization.

#### 11.2.60.2 Generalizations

VSpec

#### 11.2.60.3 Associations

- defaulValue : ValueSpecification [0..1]
  The default value of this Variable.
- replacementFragmentType {subsets type} : ReplacementFragmentType [0..1]
  The optional ReplacementFragmentType.
- type : Variabletype [1..1]
  The type of the variable.

### 11.2.61 VariableDerivation

#### 11.2.61.1 Description

A variable derivation is a specification how to derive the value of a particular variable from an expression over VSpecs.

#### 11.2.61.2 Generalizations

VSpecDerivation

#### 11.2.61.3 Associations

- derivedVariable : Variable [1..1]
  The variable it is derived from.
- derivingExpression : BCLExpression [1..1]
  The expression utilized for derivation.

## 11.2.62 Variabletype

#### 11.2.62.1 Description

The type of a variable or a value specification.

## 11.2.63 VariableValueAssignment

#### 11.2.63.1 Description

A VariableValueAssignment is a VSpecResolution which resolves a variables by providing a value of the variable's type.

#### 11.2.63.2 Generalizations

VSpecResolution

#### 11.2.63.3 Associations

- resolvedVariable : Variable [1..1]
  The resolved variable.
- value : ValueSpecification [1..1]
  The value assigned.

#### 11.2.63.4 Semantics

Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a VariableValueAssignment, the variable stored in the VariableValueAssignment must exist in the variable dictionary.
Post-condition:
After a VariableValueAssignment has been executed, the resolvedVariable is assigned with the value.
OCL :
context VariableValueAssignment::eval(ctx : CVLExecutionContext) :
pre: not ctx.variabledictionary->exists(p | p.elements->exists ( k | k.key == (self.resolvedVariable.name)))
post: ctx.variabledictionary->exists(p | p.elements->exists ( k| k.key == (self.resolvedVariable.name) and k.value == (self.value)) )

## 11.2.64 VariationPoint

### 11.2.64.1 Description

A variation points is a specification of concrete variability in the base model. Variation points define specific modifications to be applied to the base model during materialization. They refer to base model elements via base model handles and are bound to VSpecs. Binding a variation point to a VSpec means that the application of the variation point to the base model during materialization depends on the resolution for the VSpec. The nature of the dependency is specific to the kind of variation point.

### 11.2.64.2 Generalizations

VPackageable

### 11.2.64.3 Associations

- bindingVSpec : VSpec [1..1]
  The VSpecs to which the variation point is bound.

## 11.2.65 VClassifier

### 11.2.65.1 Description

A VClassifier (variability classifier) is a VSpec whose resolution requires instantiating it zero or more times and then resolving its sub-tree for each instance. When a repeatable variation point is bound to a VClassifier it will be applied once for each instance of the VClassifier during materialization.Each variability classifier has an instance multiplicity which specifies lower and upper limits for the number of instances created from it.

### 11.2.65.2 Generalizations

VSpec

### 11.2.65.3 Associations

- instanceMultiplicity : MultiplicityInterval [1..1]
  Specifies a cardinality constraint on the number of instances created from this VClassifier.

## 11.2.66 VConfiguration

### 11.2.66.1 Description

A VConfiguration is a kind of VSpecResolution which resolves a CVSpec by providing resolutions to the VSpecs in its type, which is a VInterface.VConfigurations are used to resolve configurable units.

### 11.2.66.2 Generalizations

VPackageable

VSpecResolution

#### 11.2.66.3 Attributes

- isPartial : Boolean [1..1]
  Specifies if the VConfiguration is partial.

#### 11.2.66.4 Associations

- member : VSpecResolution [0..*]
  The members of the VConfiguration.
- resolvedCVSpec : CVSpec [1..1]
  The resolved CVSpec.

#### 11.2.66.5 Semantics

Invariant : The resolutions in a configuration resolve only VSpecs of its VInterface
OCL :
-- VConfiguration
-- The resolutions in a configuration resolve only VSpecs of its VInterface
def : isContained ( in vps : VSpec ) : Boolean =
self.resolvedVIRef.vInterface.vSpec->exists(v | v = vps
or
( if not (v.childVSpec->isEmpty())
then
v.childVSpec->exists (child | isContained (child) = true)
endif
)
)
context VConfiguration
inv resolve_only_vspec_in_VInterface :
self.vSpecResolution->forAll(vspecRes | isContained(vspecRes.resolvedVSpec))

### 11.2.67 VInstance

#### 11.2.67.1 Description

A VInstance is a VSpecResolution which resolves a VClassifer by instantiating it. The child resolutions of a VInstance resolve the variability for this specific instance of the VClassifier.

#### 11.2.67.2 Generalizations

VSpecResolution

#### 11.2.67.3 Associations

- type : VClassifier [1..1]
  Type of the VInstance.

#### 11.2.67.4 Semantics

Invariant :
The number of childResolutions must be between the lowerMultiplicity and the upperMultiplicity of the instanceMultiplicity if it exists.
If the instance multiplicity does not exists, we have only one child.

OCL :
-- VInstance
-- nbChildResolution_with_instanceMultiplicity :
-- The number of childResolutions must be between the lowerMultiplicity and the upperMultiplicity of the instanceMultiplicity if it exists
-- If the instance multiplicity does not exists, we have only one child.
context VInstance :
inv nbChildResolution_with_instanceMultiplicity :
if not self.type.instanceMultiplicity->isEmpty()
then (self.childResolution->size() >= self.type.instanceMultiplicity.lower and
self.type.instanceMultiplicity.lower <> -1
and self.childResolution->size() <= self.type.instanceMultiplicity.upper and
self.type.instanceMultiplicity.upper <> -1)
-- Infinite upper
or (self.type.instanceMultiplicity.lower <> -1 and self.type.instanceMultiplicity.upper == (-1) and
self.childResolution->size() >= self.type.instanceMultiplicity.lower )
-- infinite lower no sense for practical purposes
else
self.childResolution->size() == (1)
endif
Dynamic semantics
Pre and post condition
Pre-condition :
None
Post-condition:
After a VInstance has been executed, the associated VClassifier is contained in the selected VSpecs set.
OCL :
context VInstance::eval(ctx : CVLExecutionContext)
pre :
post : ctx.selectedVSpecs->includes(self.type)

## 11.2.68 VInterface

### 11.2.68.1 Description

A VInterface is a collection of VSpecs, possibly organized in tree structures, which serves to specify what it takes to materialize a configurable unit. Each Configurable unit must be bound to a CVSpec typed by a VInterface. The VInterface may be thought of as the "variability type" or "configuration type" of the configurable unit. A VInterface can be declared as the configuration type of several configurable units since several CVSpecs may be typed by the same VInterface.

### 11.2.68.2 Generalizations

VPackageable

### 11.2.68.3 Associations

- member : VSpec [0..*]
  The members of the interface.
- ownedConstraint : Constraint [0..*]
  Constraints over the VSPecs over this VInterface

## 11.2.69 VPackage

### 11.2.69.1 Description

A VPacakge (Variability Package) is the packaging mechanism of CVL.

### 11.2.69.2 Generalizations

VPackageable

### 11.2.69.3 Associations

- packageElement : VPackageable [0..*]
  Elements contained in this VPackage.

## 11.2.70 VPackageable

### 11.2.70.1 Description

A VPackageable is an element that may be owned by a package.

### 11.2.70.2 Generalizations

NamedElement

## 11.2.71 VSpec

### 11.2.71.1 Description

A VSpecs is a specification of abstract variability. VSpecs can be organized in trees structures representing implicit logical constraints on their resolutions. VSpecs can have variation points bound to them. To materialize a base model with a variability model over it, resolutions for the VSpecs must be provided. VSpecs are organized as trees, representing logical constraints and guiding the materialization process. A VSpec may optionally a group multiplicity specifying upper and lower multiplicities against its children. The meaning of this is that each positive resolution against a VSpec must have a number of positive child resolutions conforming to the multiplicity interval. A VSpec resolution is positive if it is a choice decided positively, or any classifier instantiation, or any value assignment to a variable.

### 11.2.71.2 Generalizations

VPackageable

### 11.2.71.3 Attributes

- resolutionTime : String [1..1]
  The latest life-cycle stage at which this VSpec is expected to be resolved, e.g. "Design", "Link", "Build", "PostBuild", etc. It has no semantics within CVL.

### 11.2.71.4 Associations

- child : VSpec [0..*]
  Child VSpecs of this VSpec.

- groupMultiplicity : MultiplicityInterval [0..1]
  The group multiplicity of the VSpec. If the VSpec is resolved positively and has a group multiplicity then the number of its children resolved positively must conform to the specified multiplicity interval.
- ownedVSpecDerivation : VSpecDerivation [0..*]
  The owned VSpecDerivations.

#### 11.2.71.5 Semantics

Invariant : If the VSpec has a multiplicity interval, it must have a number of children included between the lowerMultiplicity and the upperMultiplicity
OCL :
-- VSpec
-- nbChild_with_MultiplicityInterval :
--If the VSpec has a multiplicity interval, it must have a number of children included between the lowerMultiplicity and the upperMultiplicity
context VSpec :
inv nbChild_with_MultiplicityInterval :
if not self.groupMultiplicity->isEmpty()
then (self.childVSpec->size() >= self.groupMultiplicity.lower and self.groupMultiplicity.lower
-1
and self.childVSpec->size() = self.groupMultiplicity.upper and self.groupMultiplicity.upper
-1)
-- Infinite upper
or (self.groupMultiplicity.lower
-1 and self.groupMultiplicity.upper == (-1) and self.childVSpec->size() >= self.groupMultiplicity.lower )
-- Infinite lower : no sense for practical purposes
else true
endif

## 11.2.72 VSpecDerivation

#### 11.2.72.1 Description

A VSpec derivation is a specification how to derive the resolution for a particular VSpec from resolutions for other VSpecs. When a VSpec derivation is specified for a VSpec the resolution model need not specify a resolution for it as it is calculated according to the VSpec derivation.

#### 11.2.72.2 Generalizations

VPackageable

#### 11.2.72.3 Associations

- derivedVSpec : VSpec [0..1]
  The VSpec whose value is derived by this derivation.

## 11.2.73 VSpecRef

#### 11.2.73.1 Description

A constraint expression that references a VSpec. It must reference an existing VSpec.

#### 11.2.73.2 Generalizations

BCLExpression

#### 11.2.73.3 Associations

- vSpec : VSpec [1..1]
  Reference to the VSpec.

### 11.2.74 VSpecResolution

#### 11.2.74.1 Description

TEMPLATEDESCRIPTION

#### 11.2.74.2 Generalizations

VPackageable

#### 11.2.74.3 Associations

- child : VSpecResolution [0..*]
  The child resolutions of this VSpecResolution. A given VSpecResolution is interpreted in the context of its parent.
- resolvedVSpec : VSpec [1..1]
  The VSpec this VSpecResolution resolves.Due to VSpec inheritance and VClassifiers, a given VPSec may have several VSpecResolutions resolving it, where each resolution is in the context of its parent.

# 12 Mandatory Requirements of the RFP

## 12.1 Coverage

### 12.1.1 Proposals shall define a language that can express variabilities on models in any language that is defined by means of a MOF-compliant metamodel. The language shall support:

**(1)** The most common variability mechanisms, like optionality, alternatives, etc. that have been acknowledged as mature within the fields of product line modeling and feature modeling.

**(2)** Constraints on the variabilities;

**(3)** Abstraction mechanisms that support the definition and application of compound variability specifications.

**(4)** Resolutions of the variabilities, defining the set of actual choices.

This revised submission covers the well known variability mechanisms (1); it also covers constraints (2) and compound variability specifications as VSpec can have children and Configurable Units represents hierarchical units of reuse (3). Furthermore resolutions in the form of configurations are also covered (4).

### 12.1.2 The proposed language shall specify variability as a model separate from the base model on which the variabilities apply.

Our CVL submission defines a model separate from the base model.

### 12.1.3 The proposed language shall have mechanisms for relating variability specifications to those base model elements that are subject to variation. These relationship mechanisms may assume that base models are made in languages that are defined by MOF-compliant metamodels.

Our submission has defined specific ObjectHandle and LinkHandle classes that will serve as the interface between the CVL model and the base model. The details of how the ObjectHandle and LinkHandle are realized are left to the tool vendors.

### 12.1.4 The proposed language shall be defined by means of a MOF-compliant metamodel.

Our CVL has been defined by a MOF-compliant metamodel given in this document.

### 12.1.5 Proposals shall provide a non-normative demonstration of a CVL description applied to a base model in UML including profiles.

This revised submission contains examples where the base model is defined by UML. Examples have also been made in semi-real use cases with early prototypes of CVL Tool.

## 12.2 Semantics

### 12.2.1 The proposal shall define the semantics of the variability language e.g. by using QVT or other transformation languages. The execution of a variability model with specific resolutions should result in either alterations (at runtime) of an executing product (system), or materialize (by filtering or by generation) as a specific product model in the base language.

Chapter 10 shows such an approach to the semantics of CVL which can be interpreted as executable as well as declarative. An even more detailed explanation of the semantics is given in Annex A (Chapter 16).

## 12.3 Notation

### 12.3.1 Proposals shall specify the complete concrete syntax for CVL

Chapter 9 shows concrete syntax for the variability abstraction (VSpec, Constraint and Resolution).

### 12.3.2 Proposals shall demonstrate how the notation of Feature Diagrams can be integrated within the concrete syntax of the proposed language.

Annex B (Chapter 17) explains how tools can implement visualization of variation points and obtain seamless integration with base language tools. The Annex provides a couple of examples.

### 12.3.3 Proposals shall define the notation for relationships either in separate descriptions or as annotations to the base model notation.

Annex B (Chapter 17) shows examples of both separate descriptions and annotations.

# 13 Optional Requirements of the RFP

## 13.1 Interface between CVL tool and base language tool

Proposals may define a standardized interface (e.g. by using IDL [IDL]) to be realized by the base model tools to support seamless integration with tools that support the variability language.

Our revised submission does not provide any suggestions for a standard interface.

# 14 Issues to be discussed

These issues will be considered during submission evaluation. They should not be part of the proposed normative specification. (Place them in Part I of the submission.)

## 14.1 Proposals shall discuss to which degree the proposed language can be defined by other meta-metamodeling facilities than MOF.

Our submission contains no mechanisms that should be difficult to express in other meta-metamodeling languages.

# 15 Evaluation Criteria

## 15.1 To which degree the proposed language covers exactly the domain of variability mechanisms.

The proposed CVL covers those variability constructs that are well established in the product line community and academia. CVL strives to be self-contained for most cases, but realize that for special scenarios there may be need for special means. Therefore we have defined escape mechanisms like OpaqueVariationPoint and OpaqueConstraint.

No mechanisms have been added without proper motivation with real life examples.

## 15.2 The size and complexity of the language, favoring the small and simple.

CVL is well structured and each part of the language is defined mostly by one metamodel diagram. Additional diagrams are used for more focused views.

The number of significant metaclasses is rather moderate.

# ANNEXES

# 16 Annex A: More on Semantics

## 16.1 Variation point semantics example in Kermeta

In this part we exemplify semantics using Kermeta. This semantics is expressed by adding some aspects on the CVL metamodel. In a first section we present the semantics of the ChoiceResolution and then the semantics for each of the variation points, including configurable units.

### 16.1.1 Choice Resolution semantics

Variation Points are bound directly to VSpecs. The following picture presents the semantics of ChoiceResolution in Kermeta:

```
aspect class ChoiceResolution {
    method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
                // Obtain associated VariationPoint
                var choiceVariationPoint : ChoiceVariationPoint
                choiceVariationPoint := getChoiceVariationPoint(self.resolvedChoice,ctx.vpackChoice)

                // eval Variation Point
                ctx.decision := self.decision
                if (choiceVariationPoint != void) then
                    choiceVariationPoint.eval(ctx)
                end

                // eval others choice resolutions
                if self.childResolution.size() > 0
                then
                    self.childResolution.each{ child | child.eval(ctx)}
                end
    end


    /** Retrieve the variationPoint that reference the given choice */
    operation getChoiceVariationPoint (choice : Choice,vpackChoice : VPackage  ) : ChoiceVariationPoint is do
    var choiceVariationPoint : ChoiceVariationPoint init void
    vpackChoice.packageElement.select{pe |pe.isInstanceOf(ChoiceVariationPoint) }.each {
    p |
     if p.asType(ChoiceVariationPoint).bindingVspec.exists{ c | c.name.equals(choice.name)}
     then
        choiceVariationPoint := p.asType(ChoiceVariationPoint)
     end
    }
    result := choiceVariationPoint
```

We retrieve indirectly the VariationPoint through the ChoiceResolution and call on it an eval() method.

## 16.1.2 Variation point semantics

### 16.1.2.1 LinkEndSubstitution semantics

Here is the LinkEndSubstitution semantics in Kermeta.

```
aspect class LinkAssignment {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if ( ctx.decision) // We assign a now end to the given link only if choice is  selected
    then
        ctx.strategy.assignNewEndToLink( self)
    end
end

}
```

### 16.1.2.2 ObjectSubstitution semantics

```
aspect class ObjectSubstitution {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
       if (ctx.decision) // We do the object subsitution only for selected choices
       then
          ctx.strategy.substituteElement(ctx.domainResource,
          strategyDerivation::PlacementReplacement.new.initialize(self.placementObject.object, self.replacementObject.object))
       end
end
}
```

#### 16.1.2.3 SlotAssignment semantics

```
aspect class SlotAssignment {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if ( ctx.decision) // We assign a new value to the given slot only if choice is  selected
    then
        ctx.strategy.assignNewValueToSlot( self)
    end
end

}
```

#### 16.1.2.4 FragmentSubstitution semantics

```
aspect class FragmentSubstitution {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if (ctx.decision) // We do the fragment substitution only for selected choices
    then

        // eval respectively  BoundaryElementBinding element (ToBinding or FromBinding)
        self.boundaryElementBinding.each {  bEltBinding |
            bEltBinding. eval( ctx )
        }


    end
end
}

// ToBinding and FromBinding inherits from this metaclass
aspect class BoundaryElementBinding {

operation eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do

end
}
```

```
aspect class ToBinding {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    // Bind ToPlacement and ToReplacement
    // toPlacement.outsideBounsdaryElement.propertyName[] = toReplacement.insideBoundaryElement[]
    ctx.fragmentSubstDeriv.updateRef(self.toPlacement.outsideBoundaryElement,self.toPlacement.propertyName,
    self.toReplacement.insideBoundaryElement)

    // Remove placement
    self.toPlacement.insideBoundaryElement.each { objectHandle |
        ctx.strategy.removePlacementElement(ctx.domainResource, objectHandle.object)
    }

end

}

aspect class FromBinding {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    // Update new references on replacement
    ctx.fragmentSubstDeriv.updateRef(self.fromReplacement.insideBoundaryElement,self.fromReplacement.propertyName,
    self.fromPlacement.outsideBoundaryElement)

    // Remove old references for replacement
    self.fromReplacement.eval( ctx  )

end
}

// FromReplacement inherits from this metaclass
aspect class ReplacementBoundaryElement {

operation eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
end

}

aspect class FromReplacement {

    method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
        // This call remove old reference for replacement
        ctx.fragmentSubstDeriv.removeOldReplacementRef(self )
    end

}
```

#### 16.1.2.5 ObjectExistence semantics

```
aspect class ObjectExistence {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if (not ctx.decision) // We remove element pointed by ObjectExistence only if choice is not selected
    then
        ctx.strategy.removeElement(ctx.domainResource, self.optionalObject.object)
    end
end
}
```

#### 16.1.2.6 SlotValueExistence semantics

```
aspect class SlotValueExistence {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if (not ctx.decision) // We remove the link pointed by LinkExistence only if choice is not selected
    then
        ctx.strategy.removeSlotElement(ctx, self)
    end
end

}
```

#### 16.1.2.7 LinkExistence semantics

The following figure presents the semantics of the LinkExistence in Kermeta.

```
aspect class LinkExistence {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if (not ctx.decision) // We remove the link pointed by LinkExistence only if choice is not selected
    then
        ctx.strategy.removeLinkElement(ctx.domainResource, self.optionalLink)
    end
end
}
```

#### 16.1.2.8 ParametricLinkEndSubstitution semantics

The following figure presents the semantics of the ParametricLinkEndSubstitution in Kermeta.

```
aspect class ParametricLinkAssignment {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if ( ctx.decision) // We assign a now end to the given link stored in a variable only if choice is  selected
    then
        ctx.strategy.assignNewEndToLinkParam( self)
    end
end

}
```

#### 16.1.2.9 ParametricSlotAssignment semantics

The following figure presents the semantics of the ParametricSlotAssignment in Kermeta.

```
aspect class ParametricSlotAssignmet {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if ( ctx.decision) // We assign a given slot with a new value stored in a variable only if choice is  selected
    then
        ctx.strategy.assignParamVar( self)
    end
end

}
```

#### 16.1.2.10 ParametricObjectSubstitution semantics

```
aspect class ParametricObjectSubstitution {

method eval( ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
    if ( ctx.decision) // We substitute with a new object stored in a variable only if the choice is  selected
    then
        ctx.strategy.parametricObjectSubstitution( self)
    end
end

}
```

#### 16.1.2.11 OpaqueVariationPoint semantics

```
aspect class OpaqueVariationPoint {

method eval(ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
        // We use here an unknown algorithm provided by the user
        ctx.opaqueStrategy.customVariation(ctx )
end

}
```

### 16.1.3 Configurable Unit semantics

This part presents the Configurable Unit and Configurable Unit usage semantics. To start with, we present the VConfiguration semantics called through the Configurable Unit.

#### 16.1.3.1 Configurable Unit semantics

Following is the CU aspect in which we eval the associated VConfiguration and VSpecResolution.

```
aspect class ConfigurableUnit {

method eval(ctx : CVLExecutionContext::CVLExecutionContext) : Void
    // Method eval code
    is do
            // Eval associated VConfiguration
            var vConfig : VConfiguration
                init getVConfiguration(self.bindingCVSpec,ctx.vpackChoice )
            vConfig.eval(ctx)

            // Eval all ownedVSpec Resolution
            self.ownedVSpecResolution.each { vSpecRes |
                vSpecRes.eval(ctx)
            }
    end
}
```

#### 16.1.3.1 VConfiguration semantics

The execution of a VConfiguration needs to respect an invariant, which is: for each VConfiguration inside a given VConfiguration, every contained VSpecResolution must be linked to a VSpec that is contained in the linked VInterface. Then, we evaluate all of the contained VSpecResolution.

```
aspect class VConfiguration {

//For each VConfiguration inside a given Vconfiguration, every contained VSpecResolution
//must link to a VSpec that is contained

inv respectsEncapsulation is do
    self.VConfiguration.each{ vConf |
        vConf.VSpecResolution.each{ vSpecResolution |
            vConf.resolvedVIRef.VInterface.VSpec.contains(vSpecResolution)
        }
    }
end

method eval(ctx : CVLExecutionContext::CVLExecutionContext ) : Void

// Eval method code
is do
    self.vSpecResolution.each { vSpecRes  | vSpecRes.eval(ctx)}
end
}
```

#### 16.1.3.2 CVSpecDerivation semantics

Following we present the semantics for the Composite VSpecDerivation, in which we calculate, if possible, the constraints and then evaluate the derived elements.

```
aspect class CVSpecDerivation {
    method eval (ctx : CVLExecutionContext::CVLExecutionContext) : Void is do
        //If there is a resolution for this CVSpecDerivation, it is possible to calculate the contraints
        if( vpackResolution.exists{vsr | vsr.resolvedVSpec.equals(self)}) then
            //calculate constraints
            self.memberDerivation.select{member | member.isInstanceOf(ChoiceDerivation) or
                            member.isInstanceOf(VariableDerivation)}.each{vSpecDeriv | vSpecDeriv.eval(ctx)}
            //evaluate derived CVSpecDerivations
            self.memberDerivation.select{member |
                            member.isInstanceOf(CVSpecDerivation)}.each{vSpecDeriv | vSpecDeriv.eval(ctx)}
        end
    end
}
```

## 16.2 Mapping CVL to Class Models

#### 16.2.1.1 Introduction

This section describes how to map CVL with OCL constraints to class models. CVL models are tree-like structures of Boolean choices (decisions), classifiers and variables. OCL allows for specifying constraints over the three types of variability specifications. The language, however, cannot be used as is, since semantics of CVL models and class models are different. Class models are composed of classes, attributes and associations. Classes with attributes are *flat*, i.e. Boolean attributes cannot contain other attributes or classes. CVL choices, however, can contain other choices, classifiers and variables.
We aligned OCL with tree-like structures by providing a translation from CVL to class models. The translation converts CVL models to class models, and updates OCL constraints attached to CVL models to match the structure of class models. Thus, the translation gives precise semantics to OCL constraints specified within CVL models.
From user's point of view, the class model and its semantics are hidden. User shall not be aware of the flattened tree hierarchy encoded in class models. In case of interaction (e.g. debugging, error reporting), an inverse function would project back the class model to CVL model.

#### 16.2.1.2 Informal Description

We start with an informal description of CVL models, class models, and translation rules that transform CVL to class models.

Figure 102 (a) depicts a CVL model. It is a tree-like structure of variability specifications (*choices*, *variables*,

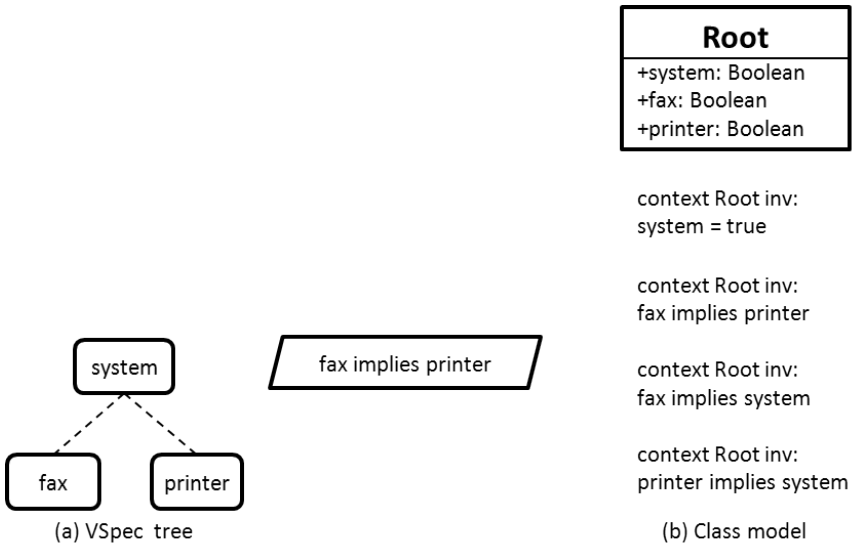



**Figure 102 Propositional Constraint**

and *classifiers*), hence its name VSpec tree. The tree starts from the *mandatory* **system** choice that has two *optional* subchoices: **fax** and **printer**. There is also an OCL *constraint* specified globally (the constraint is not nested under any VSpec), stating that fax functionality requires a printer.

Figure 102 (b) shows corresponding class model with OCL constraints. CVL choices are translated to Boolean *attributes*, as they are yes/no decisions. In class models attributes must belong to a *class*, therefore we always introduce the top-level **Root** class. In the example it contains **system**. The two other choices (**fax** and **printer**) cannot be nested under **system** in the class model. Instead we flatten the choice hierarchy, and add them as **Root**'s attributes. The *constraint* **fax implies system** is rewritten as is, but its *context* changed from global to **Root**. Finally, the hierarchical structure of VSpecs implicitly requires parent to be selected (**system**) if its child is selected (e.g. **fax**). In class models, we need to explicitly specify those constraints, such as **fax implies system**.

The above example showed only certain aspects of CVL models, i.e. simple constraints without classifiers. We are ready, however, to informally present some rules that convert a VSpec tree *M* to its class model representation *M'*. We assume that all names within the VSpec tree are unique. Later, we explain how to guarantee name uniqueness. The translation rules are as follows:

##### *16.2.1.2.1 Synthetic Root*

- Class model *M'* introduces a synthetic **Root** class to accommodate top-level CVL choices.

##### *16.2.1.2.2 Choices and Variables*

- If *c* is a choice or a variable nested under classifier *v* in *M*, then it becomes *v*'s Boolean attribute in *M'*.
- If in $c_1$, …, $c_n$ are choices or variables nested one under another, then they become attributes of $c_1$'s owner in *M'*. Thus the structure of VSpecs is *flattened*. Having **Root** in the class model guarantees that $c_1$'s owner is always a class.

#### *16.2.1.2.3 Classifiers*

- Classifier names from *M* are mapped to class names in *M'*. Furthermore, Classifier names are mapped to role names pointing to classes of the same name. The other end of association is named **parent** to allow for upwards navigation in the tree structure. Map classifier multiplicities from *M* to association multiplicities in *M'*.
- If *v* is a classifier nested under choices or variables $c_1, \ldots, c_n$, then it becomes a child of $c_1$'s owner in *M'*.

#### *16.2.1.2.4 Constraints*

- Hierarchy-preserving constraints. If choice *c* is mandatory, then add a constraint in the context of the class that owns *c* in *M'*. If *c* is mandatory, then the constraint states that *c* present whenever its parent from *M* is present (logical equivalence). For optional choices it states that if *c* is present then its parent must be present (logical implication). For example, in Figure 102 (b) **system** is mandatory, and therefore must always be present (indicated by **system = true**).
- Adjust constraints to match the class model structure in *M'*. Some constraints need rewriting since the class structure is *flat* compared to VSpec tree. Furthermore, constraints specified in context of choices in the VSpec tree are moved to classes that own corresponding attributes in the class model.
- Add constraints corresponding to group multiplicities.

### 16.2.1.3 CVL Model

To precisely define the translation between CVL and class models, we need to formally define the structure of each. A CVL model (VSpec tree) is a tuple *M* that consists of the following fields:

- $\text{VSPEC} \subseteq N_V$

  Set of unique names of VSpecs. We assume that there is a set of finite, non-empty names $N_V$. The VSPEC set is partitioned into CHOICEs and CLASSIFIERs, i.e. $\text{VSPEC} = \text{CHOICE} \cup \text{CLASSIFIER}$ and $\text{CHOICE} \cap \text{CLASSIFIER} = \emptyset$.
- $childSpec : \text{VSPEC} \rightarrow P(\text{VSPEC})$

  Function relates each VSpec with all its VSpec children. That way VSpecs form a hierarchical structure.
- $parent : \text{VSPEC} \rightarrow \text{VSPEC}$

  Partial function defined as: $\forall_{c \in childSpec(v)} parent(c) = v$. A VSpec *v* is parent of each VSpec belonging to its *childSpec* set.
- $mandatory : \text{VSPEC} \rightarrow Boolean$

  Function decides whether a VSpec is mandatory (returns *true*) or optional (returns *false*). Its value is irrelevant for classifiers. For variables it always returns *true*.
- $type : \text{VSPEC} \rightarrow Type$

  Partial function that returns the type of choice or variable. For choices it always returns *Boolean.*
- $instanceMultiplicity : \text{CLASSIFIER} \rightarrow P(N_0)$

  Function specifies the numbers of allowed instances of a classifier. The numbers are intervals of the set of natural numbers.
- $gMult : \text{VSPEC} \rightarrow P(N_0)$

  Group multiplicity. Partial function that for a VSpec *v* specifies the number of allowed VSpec children to be present under *v*.

- $constraints: \text{VSPEC} \rightarrow C_V$

  Function that for a given VSpec returns a set of constraints specified in its context. $C_V$ is a set of CVL constraints with OCL syntax.

#### 16.2.1.4 Constraints

##### *16.2.1.4.1 Concrete Syntax*

##### *16.2.1.4.2 Abstract Syntax*

#### 16.2.1.5 Example

The following tuple represents VSpec tree from Figure 102 (a).

*M* = (

| | |
|---|---|
| VSPEC | = {system, fax, printer}, |
| *childSpec* | = {(system, {fax, printer}), (fax, {}), (printer, {})}, |
| *parent* | = {(fax, system), (printer, system)}, |
| *mandatory* | = {(system, *true*), (fax, *false*), (printer, *false*)}, |
| *type* | = {(system, Boolean), (fax, Boolean), (printer, Boolean)}, |
| *instanceMultiplicity* | = {}, |
| *gMult* | = {}, |
| *constraints* | = {**context system inv: fax implies printer**} |

)

#### 16.2.1.6 Constrained Class Model

The structure of constrained class model is also a tuple. Class models can be arbitrary graphs, but we use them to encode tree-like VSpec structure *M′*. The tuple *M′* is composed of the following fields:

- $\text{CLASS} \subseteq N_C$

  Set of unique names of classes. We assume that there is a set of finite, non-empty names $N_C$.

- $\text{ATT} = \bigcup_{c \in \text{CLASS}} \text{ATT}_c$

  Set of attributes defined as a union of attributes belonging to each class. Attributes of a class *c* are defined as a set $\text{ATT}_c$ of functions $a \rightarrow (\text{Class} \rightarrow Type)$, where the attribute name $a \in N_C$; $Type$ is a set of types.

- OP

  Set of operations associated with classes. Although it is a part of class model, there are no operations in CVL models.

- $\text{ASSOC} \subseteq N_C$

  Finite set of association names. All association names are unique. We use binary associations to construct tree hierarchy among classes. We assume that all associations mean object composition.

- $associates: \text{ASSOC} \rightarrow \text{CLASS} \times \text{CLASS}$

  Function maps each association name to a pair of classes participating in the association. The first element of the pair encodes parent, while the second its child.

- $roles: \text{ASSOC} \rightarrow N_C \times N_C$

  Function assigns each pair of classes participating in the association a pair of role names. The first element of the pair indicates parent, while the second its child. Role names are needed for navigation along the tree structure in constraints.

- $multiplicities: N_C \rightarrow P(N_0)$

  Function assigns each role name a set representing the number of allowed class instances.

- $\prec_C$

  Partial order on the set of classes. Specifies inheritance hierarchy among them. We are not concerned with inheritance among VSpecs.
- $C_C$

  Set of OCL constraints.

The tuple below defines the class model from Figure 102 (b):

*M'* = (

| | |
|---|---|
| CLASS | = {Root}, |
| ATT | = {(system, (Root, Boolean)), (fax, (Root, Boolean)), (printer, (Root, Boolean))}, |
| OP | = {}, |
| ASSOC | = {}, |
| *associates* | = {}, |
| *roles* | = {}, |
| *multiplicities* | = {}, |
| $\prec_C$ | = {}, |
| $C_C$ | = {**context Root inv: system = true, context Root inv: fax implies printer, context Root inv: fax implies system, context Root inv: printer implies system**} |

)

#### 16.2.1.7 Translation

Once the source (VSpec tree) and target (class model) structures are defined, we construct function *T* that takes VSpec tree, traverses it, and returns corresponding class model. The function is defined as:

$$T(t) = (\{\text{Root}\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset) \uplus T'(\text{Root}, \text{root}_V(t))$$

where:

$$(a_1, a_2, \dots, a_n) \uplus (b_1, b_2, \dots, b_n) = (a_1 \cup b_1, a_2 \cup b_2, \dots, a_n \cup b_n)$$

The $\uplus$ function takes two tuples of sets and returns a tuple of pairwise set unions. We also use it to make a union of functions.

Furthermore $\text{root}_V(t) : M \to \text{VSPEC}$ is a function that returns the root of CVL model (such as **system** for the model from Figure 102 (a)). The Root label indicates explicit root class added to the class model.

Function $T' : \text{CLASS} \times \text{VSPEC} \to M'$ translates a VSpec and its children to a constrained class model. If invoked with parameters $T'(c, v)$, the parameters are: *c* is the class that will own currently processed VSpec, *v* is currently processed VSpec.

Function *T'* is defined recursively:

$$T'(c, v) = \begin{cases} T_{\text{Attribute}}(c, v) \uplus \left(+\right) ch \in childspec(v) \left(T'(c, ch)\right), & \text{if } v \in \text{CHOICE} \cup \text{VARIABLE} \\ T_{\text{Classifier}}(c, v) \uplus \left(+\right) ch \in childspec(v) \left(T'(c, ch)\right), & \text{if } v \in \text{CLASSIFIER} \end{cases}$$

and dispatches on VSpec type. In all cases, it constructs a class model *M'* using $T_{\text{Attribute}}$, or $T_{\text{Classifier}}$ function. Then, it merges the model with a union of class models generated for children of *v*. For now assume that names are unique – we will come back to this later. We use the following helper functions:

- $T_{\text{Attribute}} : \text{CLASS} \times (\text{CHOICE} \cup \text{VARIABLE}) \to M'$

  $$T_{\text{Attribute}}(c, v) = \left(\emptyset, \{v \to (c \to type(v))\}, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, mkConstraints(c, v)\right)$$

  Takes class *c* and choice or variable *v* and makes it an attribute of *c*. If necessary, adds extra constraints.
- $T_{\text{Classifier}} : \text{CLASS} \times \text{CLASSIFIER} \to M'$

$$T_{\text{Classifier}}(c,v) = (\{v\}, \emptyset, \emptyset, \{v\},$$
$$\{v \to (c,v)\},$$
$$\{v \to (parent,v)\},$$
$$\{v \to (\{1\}, instanceMultiplicity(v))\},$$
$$mkConstraints(v,v))$$

Takes class *c* and classifier *v*, and translates it to class model. The function creates new class and association, both named *v*. Then defines the association between class *c* and *v*, and establishes role names *parent* and *v*, for *c* and *v* respectively. For each role name defines the number of allowed instances. Finally, adds extra constraints to the new class *v*.

- $mkConstraints:$ CLASS × VSPEC → $C_C$

  $mkConstraints(c,v) = addDependencies(c,v) \cup mkGMult(c,v) \cup T_{Constr}(c,v)$

  Translates existing CVL constraints so that they match the class model, and also generates extra OCL constraints. Extra constraints include dependencies between a VSpec and its parent, and group multiplicities.
- $addDependencies:$ CLASS × VSPEC → $C_C$

  $addDependencies(c,v)$

  $$= \begin{cases} \{\text{context } c \text{ inv: } \varphi_{self}(v) \oplus \varphi_{op}(v) \oplus parent(v)\}, & \text{if } parent(v) \in \text{CHOICE} \wedge v \notin \text{VARIABLE} \\ \{\text{context } c \text{ inv: } v = true\}, & \text{if } parent(v) \in \text{CLASSIFIER} \wedge \\ & mandatory(v) \wedge v \in \text{CHOICE} \\ \emptyset, & \text{otherwise} \end{cases}$$

  $$\varphi_{self}(v) = \begin{cases} v, & \text{if } v \in \text{CHOICE} \\ v \to \text{nonEmpty}(\ ), & \text{if } v \in \text{CLASSIFIER} \end{cases}$$

  $$\varphi_{op}(v) = \begin{cases} =, & \text{if } mandatory(v) \\ \text{implies}, & \text{otherwise} \end{cases}$$

  $addDependencies$ takes class *c* and VSpec *v*, and specifies dependencies between *v* and its parent in the context of *c*. Dependencies come from the tree-like structure of CVL models.
- $\oplus : String \times String \to String$

  Concatenates two strings.
- $mkGMult:$ CLASS × VSPEC → $C_C$

  $$mkGMult(c,v) = \begin{cases} \emptyset, & \text{if } gMult(v) = \perp \\ \{\text{context } c \text{ inv: let n} = count(childSpec(v)), & \text{otherwise} \\ \text{in } min(gMult(v)) \leq \text{n} \oplus boundary(v)\} & \end{cases}$$

  Takes class *c* and VSpec *v*, and generates group multiplicity constraints for *v*. The constraints are specified in the context of *c* and restrict the number of allowed children. The function assumes that functions *min* and *max* take subsets of natural numbers and return their minimum (maximum).
- $count : P(\text{VSPEC}) \to String$

$$count(vs) = \begin{cases} (v\,?\,1:0) + \bigoplus count(vs \setminus v), & \text{if } \exists v \in vs \wedge v \in \text{CHOICE} \\ 1 + \bigoplus count(vs \setminus v), & \text{if } \exists v \in vs \wedge v \in \text{VARIABLE} \\ v \to size(\quad) + \bigoplus count(vs \setminus v), & \text{if } \exists v \in vs \wedge v \in \text{CLASSIFIER} \\ 0, & \text{if } vs = \emptyset \end{cases}$$

The function generates formulas for counting the actual VSpec multiplicities. CVL choices are Boolean attributes, so they are either counted as one (if present) or zero (otherwise). For classifiers, it returns OCL's **size** function that counts the actual number of class instances.

- $boundary : \text{VSPEC} \to String$

$$boundary(v) = \begin{cases} \epsilon, & \text{if } |gMult(v)| = \aleph_0 \\ \text{and n} \leq max\big(gMult(v)\big), & \text{otherwise} \end{cases}$$

For given VSpec determines whether there is an upper limit on group multiplicity. If there is none, then returns an empty string. Otherwise adds a part of constraint that sets this limit on contained children.

- $T_{Constr} : \text{CLASS} \times \text{VSPEC} \to C_C$

Translates VSpec constraints to OCL constraints attached to class model. We are not going to specify this function fully here. It copies existing constraints with the following modifications:

  - $T_{Context}(\text{context } v \text{ inv : } \psi) = \text{context } c \text{ inv : } v \text{ implies } \psi$

    Updates context of the constraint so that it is always attached to a class. Class models cannot have constraints specified in the context of attributes (as opposed to constraints in the context of CVL choices). Class *c* owns *v* in *M'*. Change of the context requires that if a formula $\psi$ is specified under choice *v* in *T*, the constraint is updated to implication.

    That way, the constraint must only hold when the choice is present.

  - $T_{Nav}(\alpha) = T_{Flatten}(T_{Parent}(\alpha))$

    Rewrites navigation expressions to match the flattened hierarchy of nested choices. First, it resolves parent of each VSpec for which **parent** keyword appears in the constraint.

- $T_{Parent} : String \to String$

$$T_{Parent}(v.\alpha.\beta) = \begin{cases} T_{Parent}(v.\gamma.\beta), & \text{if } \alpha = \gamma.\text{w.parent} \\ T_{Parent}(v \oplus mkPath(v).\beta), & \text{if } \alpha = \text{parent} \wedge v \in \text{CHOICE} \cup \text{VARIABLE} \\ \text{v.parent} \oplus T_{Parent}(mkPath(v).\beta), & \text{if } \alpha = \text{parent} \wedge v \in \text{CLASSIFIER} \\ v.\alpha.\beta, & \text{otherwise} \end{cases}$$

CVL constraints may involve navigation to parent VSpec by using **parent** function. For example, the following expression $c_1.c_2 \ldots c_{n-1}.c_n.parent$ would point to $c_{n-1}$. $T_{Parent}$ resolves the actual parent for each VSpec and rewrites the expression to match the class model. If navigation goes from a VSpec to its parent, the VSpec and its parent are removed from the expression. If the parent navigation follows a choice, the function removes parent from the expression, since the choice belongs to a classifier. If the parent navigation follows a classifier, the function navigates to parent by using the role name and then navigates down if *v*'s parent is a choice. In all other cases, copies the expression.

- $mkPath : \text{VSPEC} \to String$

$$mkPath(v) = \begin{cases} mkPath\big(parent(v)\big).parent(v), & \text{if } parent(v) \in \text{CHOICE} \cup \text{VARIABLE} \\ \epsilon, & \text{otherwise} \end{cases}$$

For given VSpec the function generates path from root to the VSpec's parent.

- $T_{Flatten}(\alpha.\mathrm{a}) = T'_{Flatten}(\alpha).\mathrm{a}$, where $\alpha \in$ VSPEC

| | | |
|---|---|---|
| $T_{Flatten}(\mathrm{self})$ | = self | |
| $T_{Flatten}(\mathrm{x})$ | = x | where $x \in$ CLASSIFIER |
| $T_{Flatten}(\alpha.\mathrm{x})$ | = $T_{Flatten}(\alpha).\mathrm{x}$ | where $x \in$ CLASSIFIER |
| $T_{Flatten}(\alpha.\mathrm{c}_1.\mathrm{c}_2 \ldots \mathrm{c}_n)$ | = $T_{Flatten}(\alpha)$ | where $\mathrm{c}_i \in$ CHOICE ∪ VARIABLE |

Rewrites navigation expressions to match the flattened class hierarchy. For choice paths (the last case) removes the whole path since all those choices are nested under the same class in the class model.

#### 16.2.1.8 Name Resolution

So far we assumed that all VSpec names are unique within the model. This assumption is valid, because all the names can be resolved and uniquely identified before translation. For each resolved name compute hash of its complete path (fully qualified name), i.e. $H(v_1.v_2 \ldots v_n)$, where *H* is a injective hash function. The hash becomes VSpec's class/attribute name in the class model. Update names used in constraints accordingly.

The mapping between fully qualified names and hashes is provided by constructing a look-up table of hashes and corresponding VSpec names. This table might be useful for translating back class model to VSpec tree, or debugging.

#### 16.2.1.9 Name Resolution Rules

Name resolution rules disambiguate names of VSpecs used in constraints. A name is resolved in context of a VSpec in up to five steps.

6. Check if it is a special name *self* or *parent*. The former indicates VSpec in which context the constraint is specified. The latter maps to the *parent* VSpec of the VSpec in which the constraint is specified. If the context is a root VSpec, so it has no parent, the constraint is not well-formed (an error).
7. Look up the name in children of the context node in breadth-first search manner.
8. Search in the VSpec's ancestors starting from the VSpec and up. For each ancestor, look up the name also in its descendants.
9. Search in other top-level definitions.
10. If the name cannot be resolved or is ambiguous within a single step, the constraint is not well-formed and an error is reported.

For navigations (expressions of the form $v_1.v_2...v_n$) the name resolution rules are applied to resolve $v_1$. Once it is resolved, subsequent VSpecs ($v_2...v_n$) are resolved by applying only rules (1), (2), and (5).

#### 16.2.1.10 Examples

This section contains extra examples. Each example has specified VSpec tree *M*, its graphical representation, corresponding class model *M'* and its visualization.

##### *16.2.1.10.1 Existential Quantification*

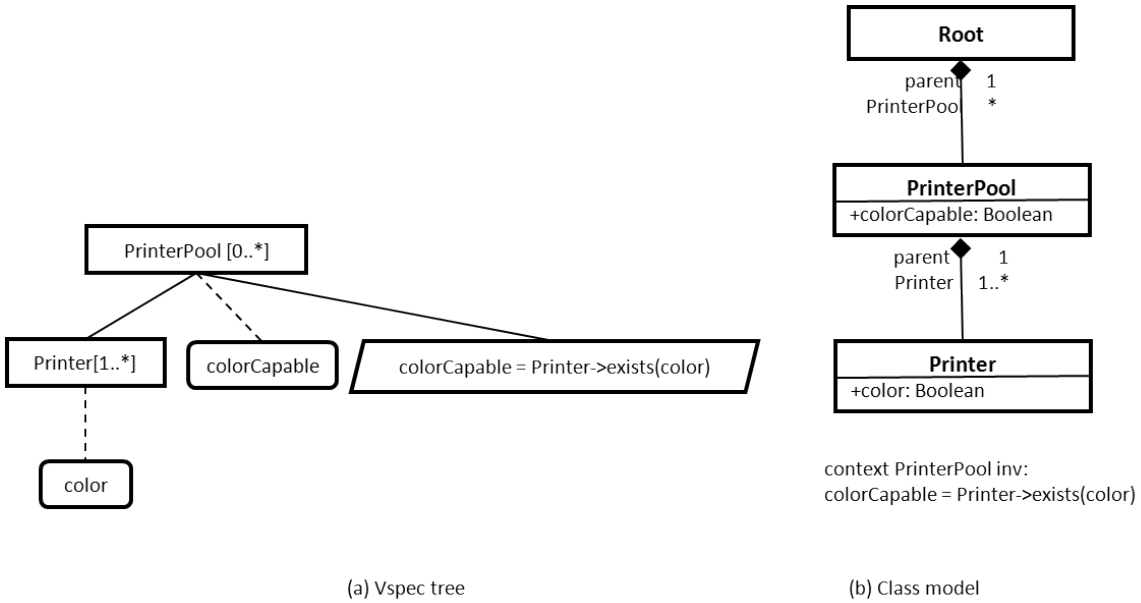



**Figure 103: Existential quantification**

$M$ = (

| | |
|---|---|
| VSPEC | = {PrinterPool, Printer, color, colorCapable}, |
| *childSpec* | = {(PrinterPool, {Printer, colorCapable}), (Printer, {color}), (color, {}), (colorCapable, {})}, |
| *parent* | = {(Printer, PrinterPool), (color, Printer), (colorCapable, PrinterPool)}, |
| *mandatory* | = {(PrinterPool, *false*), (Printer, *false*), (colorCapable, *false*), (color, *false*)}, |
| *type* | = {(colorCapable, Boolean), (color, Boolean)}, |
| *instanceMultiplicity* | = {(PrinterPool, [0,∞)), (Printer, [1, ∞))}, |
| *gMult* | = {}, |
| *constraints* | = {**context PrinterPool inv: colorCapable = Printer->exists(color)**} |

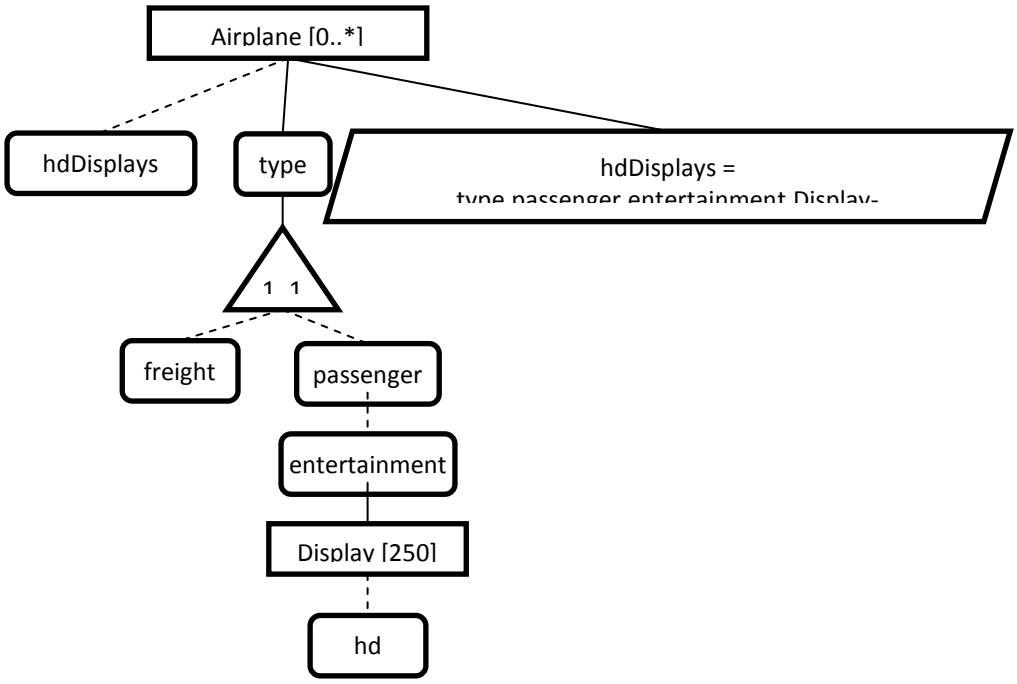
)

$M'$ = (

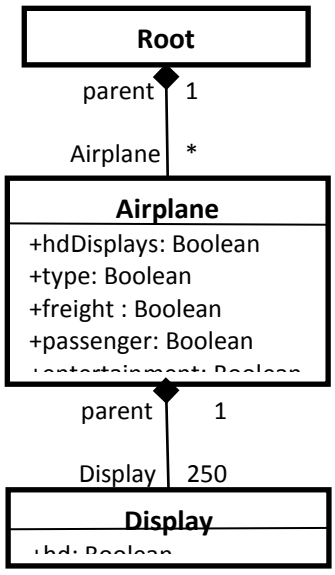
| | |
|---|---|
| CLASS | = {Root, PrinterPool, Printer}, |
| ATT | = {(color, (Printer, Boolean)), (colorCapable, (PrinterPool, Boolean))}, |
| OP | = {}, |
| ASSOC | = {PrinterPool, Printer}, |
| *associates* | = {(PrinterPool, (Root, PrinterPool)), (Printer, (PrinterPool, Printer))}, |
| *roles* | = {(PrinterPool, (parent, PrinterPool)), (Printer, (parent, Printer))}, |
| *multiplicities* | = {(PrinterPool, [0,∞)), (Printer, [1,∞))}, |
| $\prec_C$ | = {}, |
| $C_C$ | = {**context PrinterPool inv: colorCapable = Printer->exists(color)**} |

)

### *16.2.1.10.2 Nested Choices and Classifiers*




(a) Vspec tree




(b) Class model

context Airplane inv:
hdDisplays = Display->forAll(hd)

context Airplane inv:
type = true

context Airplane inv:
freight implies type

context Airplane inv:
passenger implies type

context Airplane inv:
let n = (freight ? 1 : 0) + (passenger ? 1 : 0) + 0 in 1 <= n and n <= 1

context Airplane inv:
entertainment implies passenger

context Airplane inv:
Display->notEmpty() implies entertainment

**Figure 104: Nested VSpecs**

$M =$ (

VSPEC = {Airplane, hdDisplays, type, freight, passenger, entertainment,

| | | |
|---|---|---|
| | | Display, hd}, |
| | *childSpec* | = {(Airplane, {hdDisplays, type}), (hdDisplays, {}), (hd, {}), (type, {freight, passenger}), (freight, {}), (Display, {hd}), (passenger, {entertainment}), (entertainment, {Display})}, |
| | *parent* | = {(hdDisplays, Airplane), (type, Airplane), (freight, type), (passenger, type), (entertainment, passenger), (Display, entertainment), (hd, Display)}, |
| | *mandatory* | = {(Airplane, *false*), (hdDisplays, *false*), (type, *true*), (freight, *false*), (passenger, *false*), (entertainment, *false*), (Display, *false*), (hd, i)}, |
| | *type* | = {(hdDisplays, Boolean), (type, Boolean), (freight, Boolean), (passenger, Boolean), (entertainment, Boolean), (hd, Boolean)}, |
| | *instanceMultiplicity* | = {(Airplane, [0,∞)), (Display,{250)}, |
| | *gMult* | = {}, |
| | *constraints* | = {**context Airplane inv: hdDisplays = type.passenger.entertainment.Display→forAll(hd)** } |
| | ) | |
| *M'* = | ( | |
| | CLASS | = {Root, Airplane, Display}, |
| | ATT | = {(hdDisplays, (Airplane, Boolean)), (type, (Airplane, Boolean)), (freight, (Airplane, Boolean)), (passenger, (Airplane, Boolean)), (entertainment, (Airplane, Boolean)), (hd, (Display, Boolean))}, |
| | OP | = {}, |
| | ASSOC | = {Airplane, Display}, |
| | *associates* | = {( Airplane, (Root, Airplane)), (Display, (Airplane, Display))}, |
| | *roles* | = {( Airplane, (parent, Airplane)), (Display, (parent, Display))}, |
| | *multiplicities* | = {( Airplane, [0,∞)), (Display, {250})}, |
| | $\prec_C$ | = {}, |
| | $C_C$ | = { **context Airplane inv: hdDisplays = type.passenger.entertainment.Display→forAll(hd)**} |
| | ) | |

# 17 Annex B Concrete Syntax of Variation Points

## 17.1 Variability Realization

CVL will not specify any normative concrete syntax for the variability realization, but we shall show some examples of how the realization can be visualized.

We shall show one example of how the variation points can be expressed statically by amalgamating notation with that of the base model. Our examples use UML as base language.

Our second example indicates how the realization can be visualized dynamically by color overlays.

### 17.1.1 Static variability realization through notation amalgamation

Even though CVL advocates a separate approach to variability it may be useful to apply an amalgamated approach to visualizing how the variability affects the base descriptions. Whether such a static description of variability options onto the base model is fruitful is dependent upon how the variability is realized. How well this description form functions is also dependent upon how easy it is to enhance the base language with variability notation. Our example here uses UML as the base language and shows how the VSpec

model of Figure 1 can be realized. Figure 105 indicates how the CVL realization model refers to the VSpec model and the base model.

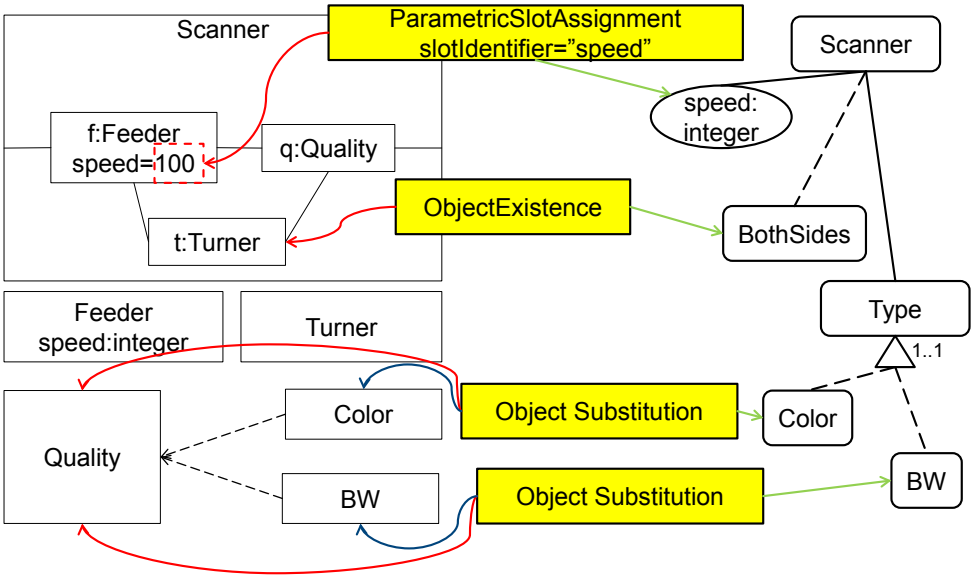



**Figure 105 Realization model for Scanner**

In Figure 106 we give one possible way to amalgamate information from the VSpec model into the base model.

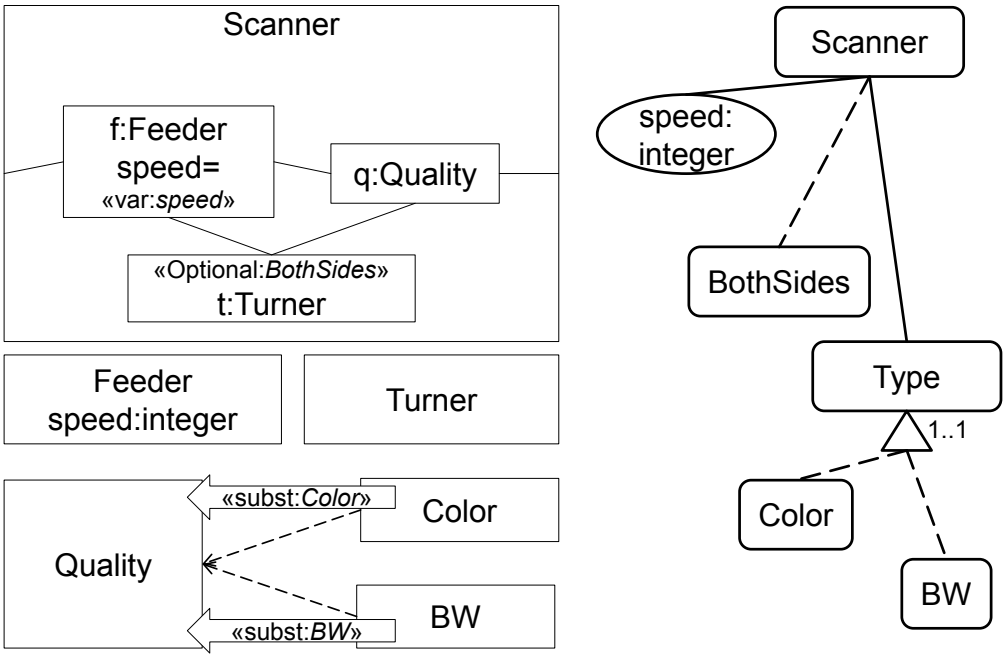



**Figure 106 The amalgamated UML with stereotype-like variability notation**

We have applied the following constructs in our example notation (which is only a possible example and not normative in any sense):

1. Stereotype-like texts relating to symbols in the base model. Here we have included inside the guillemet both a keyword indicating the corresponding CVL construct and the name of the CVL object. (Here: var:speed meaning variability value resolving the variable spee, and

*Optional:BothSides* indcating the the *t:Turner* will be present only if the choice *BothSides* are resolved to **true**.)

2. New proprietary symbol which describes a relationship between the source and the target of an object substitution. We have introduced a fat arrow containing text indicating the kind of relationship and the name of the corresponding CVL object. (Here: *subst:Color* which shows that it is an object substitution depending on the resolution of the choice *Color*. If Color is resolved to true, then the UML class *Color* will be replacing *Quality* in the *Scanner*.)

### 17.1.2 Dynamic visualization of variability realization

Static amalgamation of the notation is not always practical. Since the variability model is in principle totally separate from the base model and only referring into the base model, it is possible to have many VSpecs that refer the same base model objects. In that situation the amalgamated diagrams will become rather crowded and hard to comprehend.

For such situations we may rather apply a dynamic approach which can be seen more as a tooling trick than concrete syntax, but this depends on what we want to consider as concrete syntax.

We illustrate an approach to highlighting the variability realization with the fragment substitution example shown again in Figure 107.

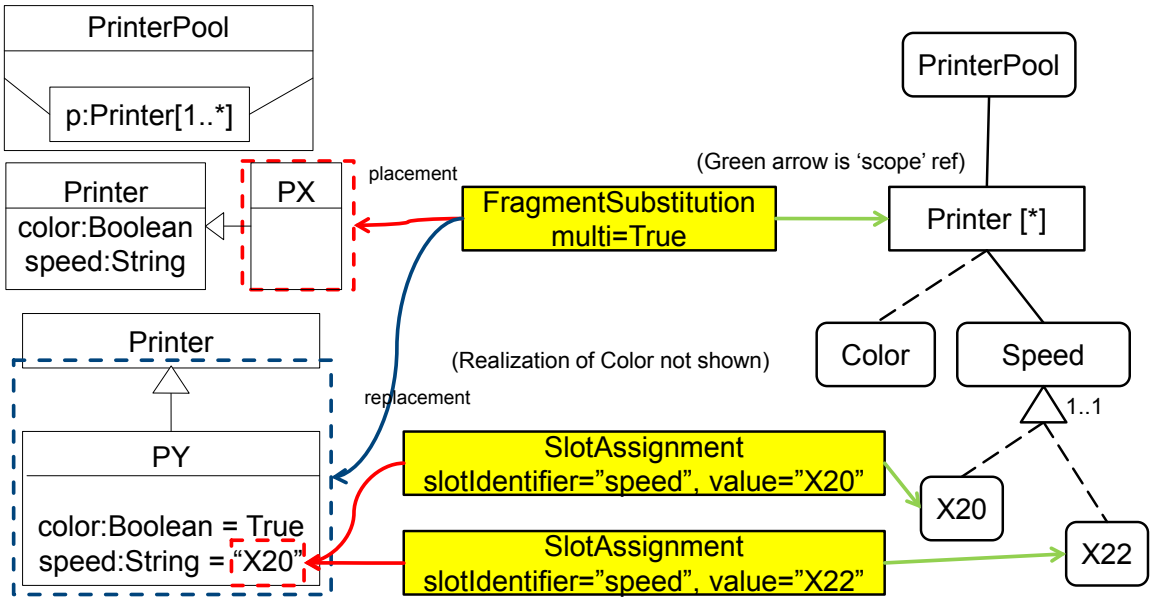



**Figure 107 Fragment Substitution realization**

In Figure 107 we see that there is some overlap between the affected areas of the base model and it is not obvious that a static amalgamation would be the most fruitful. In general a fragment contains more than one object and therefore static amalgamation would require more proprietary notation e.g. like the colored frames shown in the illustration of Figure 107.

Instead we may apply dynamic visualization which exploits a tool connection between the base model editor and the CVL tool.

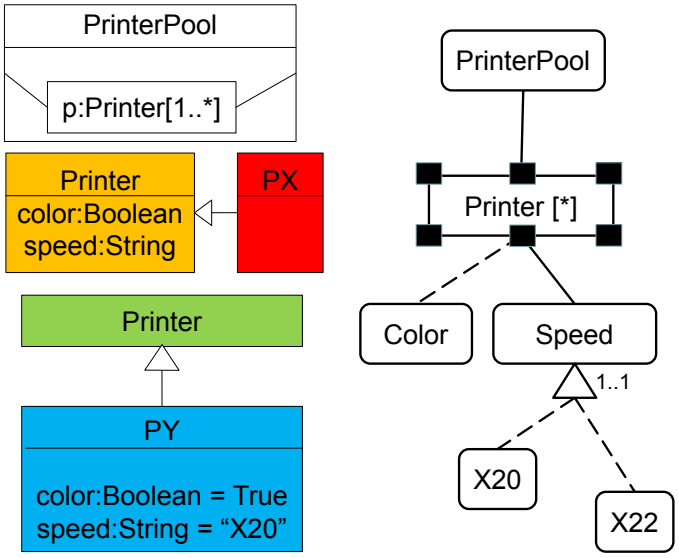



**Figure 108 Dynamic visualization of fragment substitution realizing a VClassifier**

In Figure 108 we indicate that the base model is highlighted with color overlays representing the realization of the selected (Printer) VClassifier. Red indicates what will be removed and orange what that would affect. Blue indicates what will be replacing the red, and green what that will affect.

How the tool chooses to indicate the realization is up to the tool and most probably dependent on the base language. We have in our example used color overlays, but we could have used any clearly visible change of the model. For textual syntaxes we could use different font or style (bold face, italics, underline) in much the way that we know from program language editors.