Project Number: 295397

VARIES

VARiability **I**n safety-critical **E**mbedded **S**ystems

ARTEMIS-2011-1

D4.2 BVR - The language

Start date of project: May, 1st 2012
Duration: 36 months

Deliverable	**D4.2** BVR - The language		
Confidentiality	PP	**Type**	R
Project	VARIES	**Date**	2014-06-24
Status	FINAL	**Version**	01
File Identifier	VARIES_D4.2_v01_PP_FINAL		

Contact Person	**Øystein Haugen**
Organisation[1]	SINTEF
Phone	+47 913 90 914
E-Mail	oystein.haugen@sintef.no

[1] See Appendix I (Partner identification) on page 48.

THE RESEARCH LEADING TO THESE RESULTS HAS RECEIVED FUNDING FROM THE SVENTH FRAMEWORK PROGRAMME AND ARTEMIS-JU UNDER GRANT AGREEMENT N° 295397

AUTHORS

Name	Organization1	E-Mail
Øystein Haugen	SINTEF	oystein.haugen@sintef.no

REVIEWERS
(of the current version)

Name	Organization1	E-Mail
Klaas Gadeyne	FMTC	klaas.gadeyne@fmtc.be
Andrzej Wasowski	ITU	andrzej.wasowski@itu.dk

CHANGE HISTORY

Version	Date	Reason for Change	Sections / Pages Affected
01	2014-06-24	First official version. Deliverable title has been modified in response to CVL patent litigation.	All

CONTENT

LIST OF FIGURES

LIST OF TABLES

APPENDIX

1 INTRODUCTION

1.1 OVERVIEW, PURPOSE AND SCOPE

This document describes BVR – the language that will satisfy the needs expressed by the VARIES partners. BVR builds on CVL, but BVR is not a pure superset of CVL.

The goal of this document is to give the language definition produced from the formal metamodel which will in turn be used to create a BVR Tool Bundle.

For references to the work plan, see [VARIES-TA], B3.3.

1.2 PARTNER INVOLVEMENT

Table 1. Partner involvement

Partner	Involvement
SINTEF	Major creator and editor
ITU	Discussions and cooperation
Autronica	Discussions and review
Fraunhofer	Production of hyperlinked manual from metamodel

2 BVR Metamodel Diagrams

2.1 Variability Abstraction

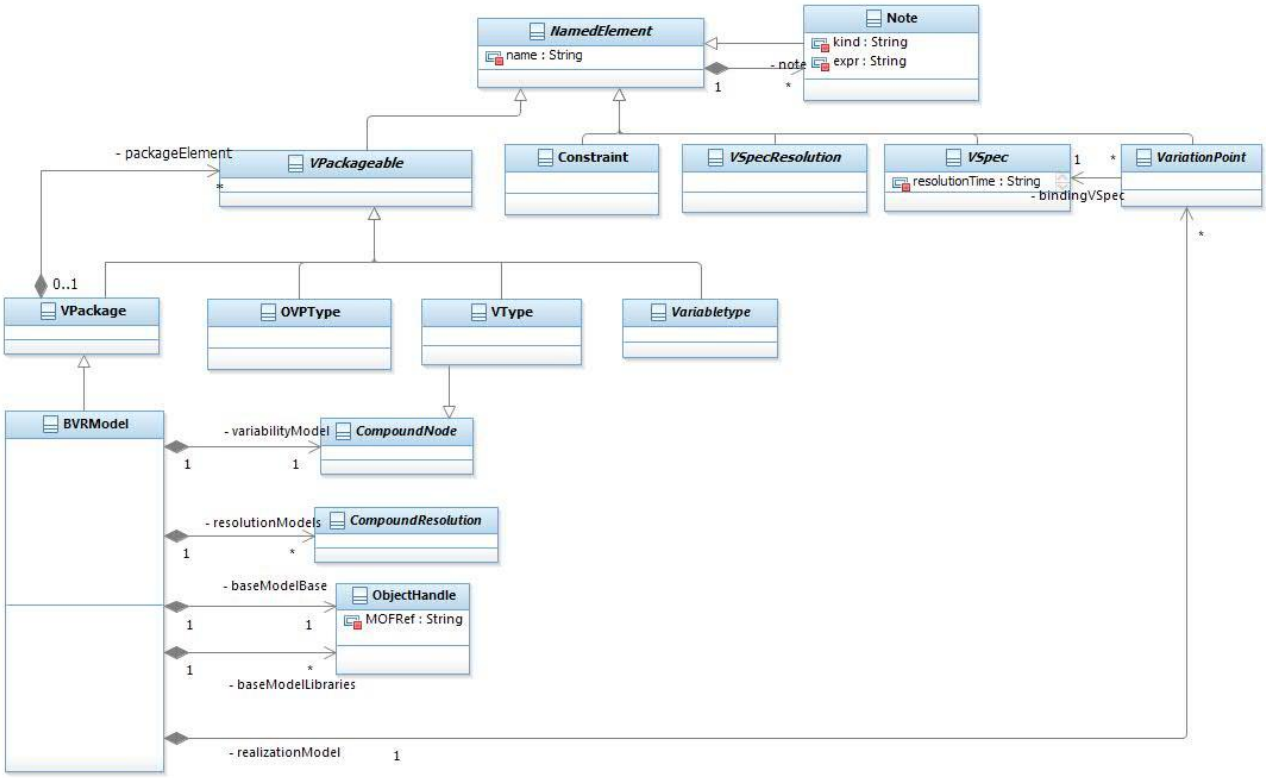

Figure 1 BVR top level

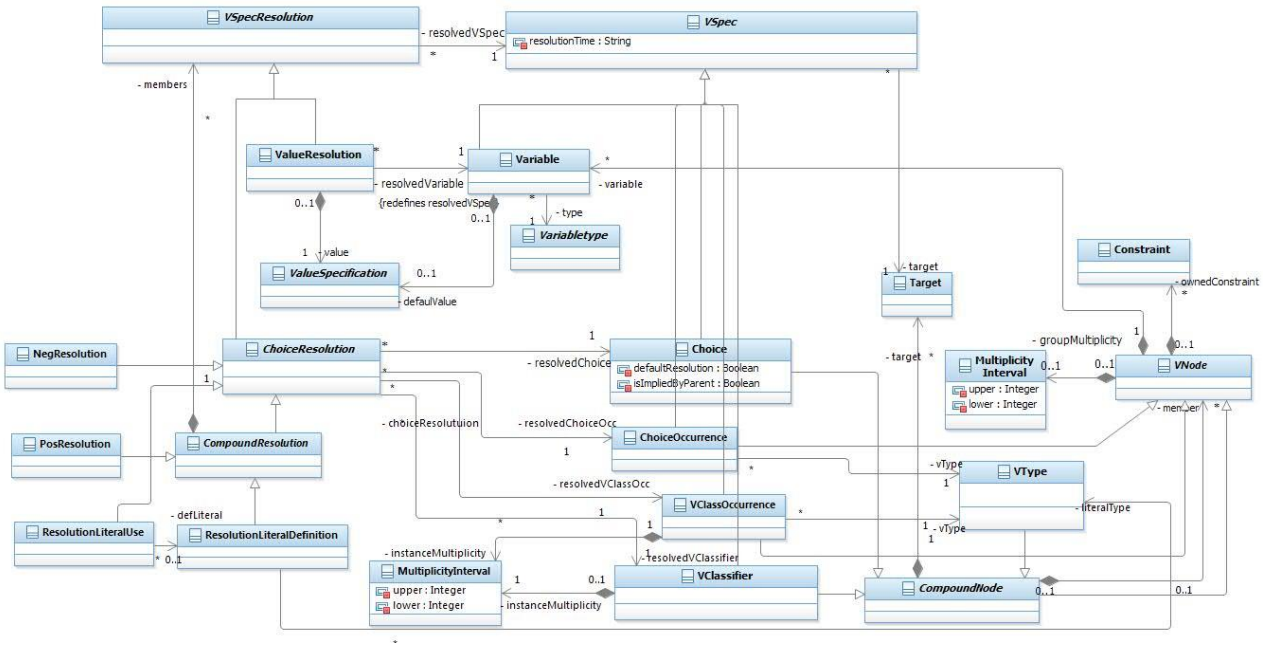

Figure 2 VSpec and VSpecResolutions

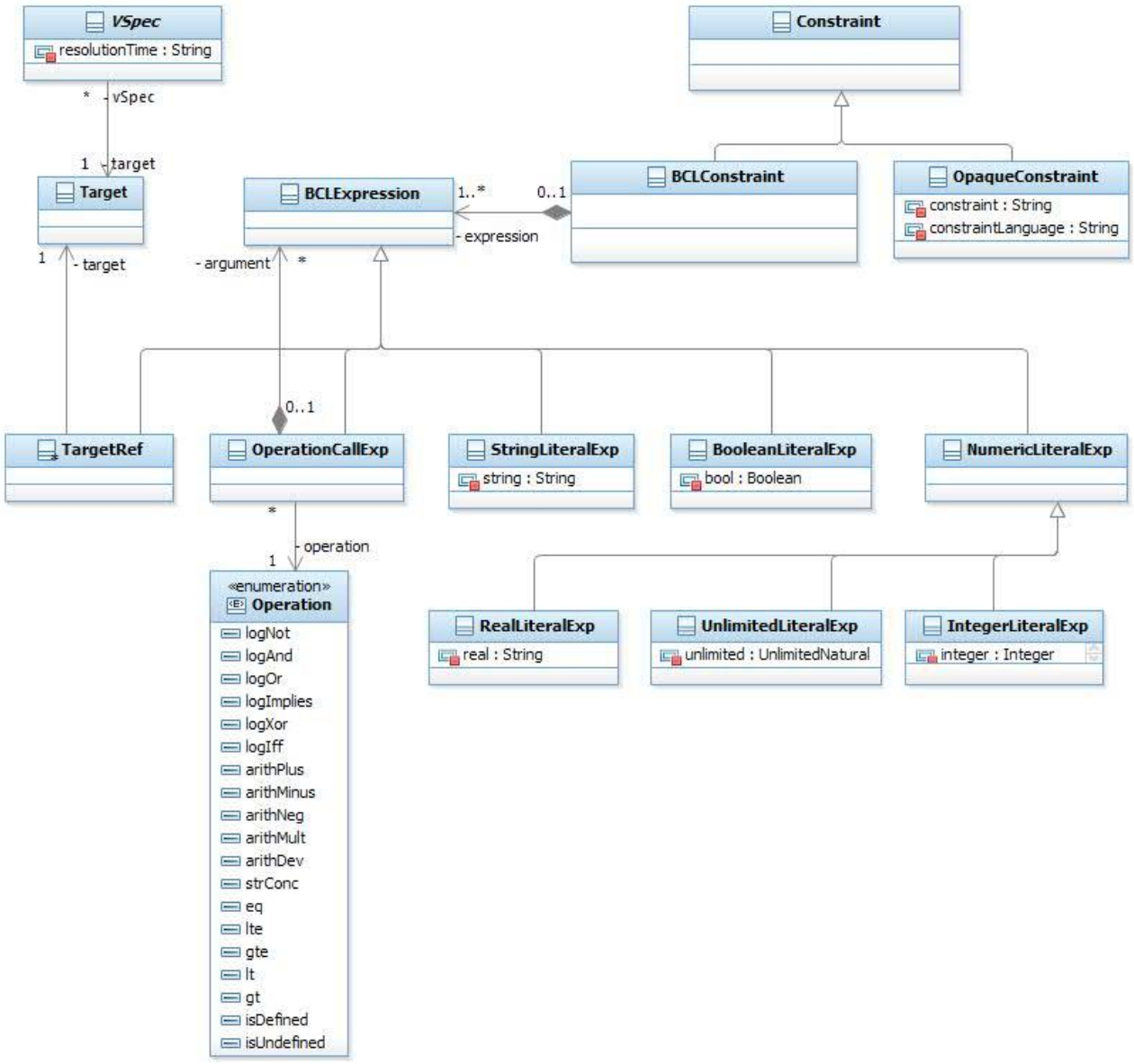

Figure 3 Constraints

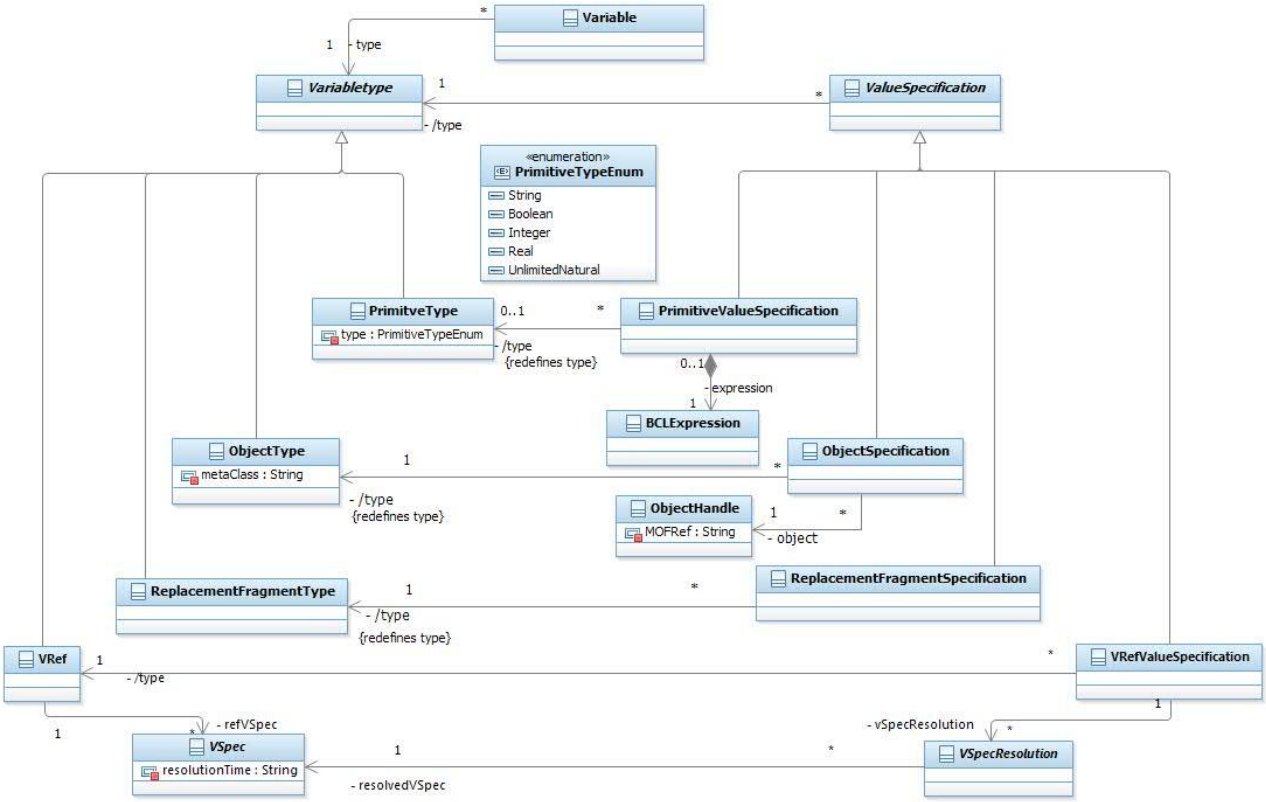

Figure 4 Variable and the associated types

2.2 REALIZATION

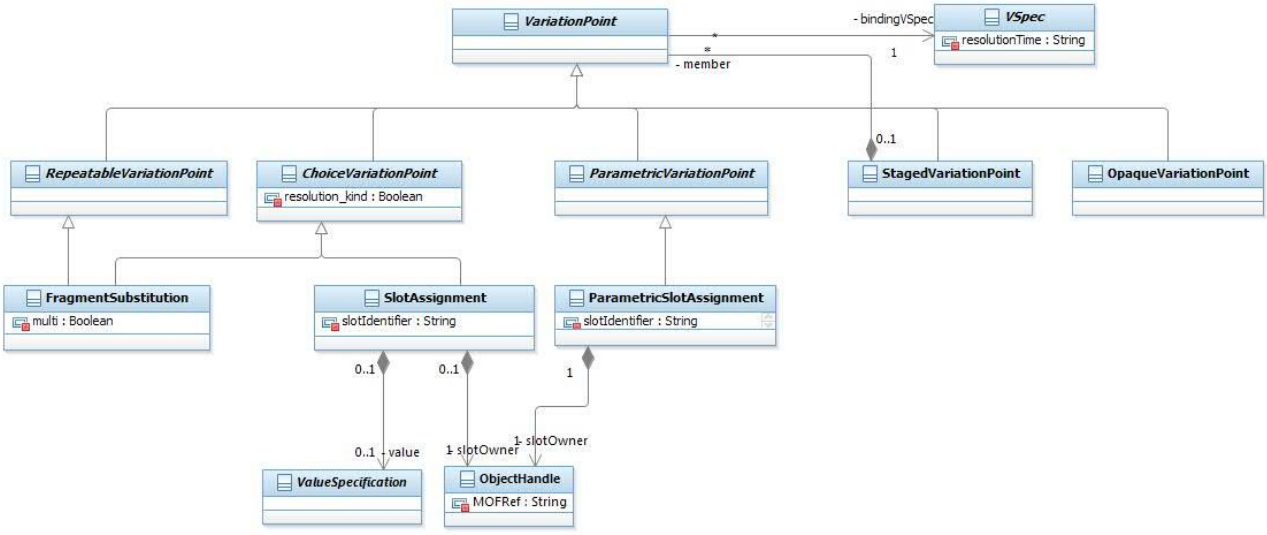

Figure 5 Variation Point taxonomy

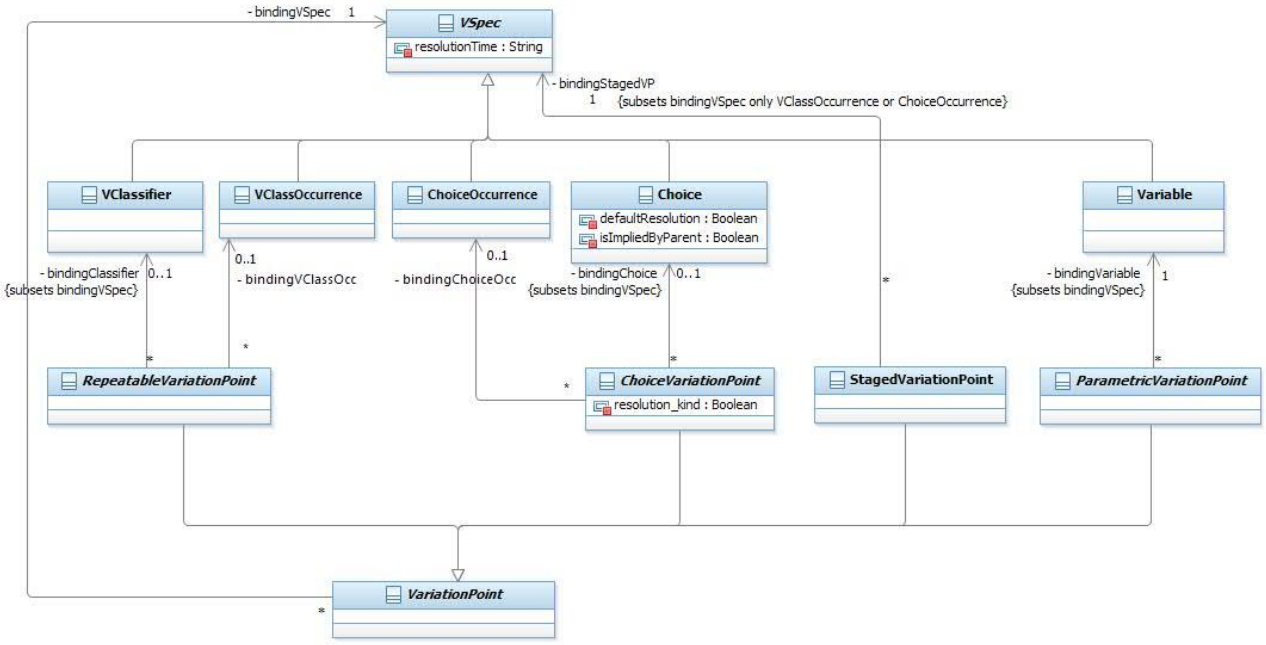

Figure 6 Variation Point to VSpec mapping

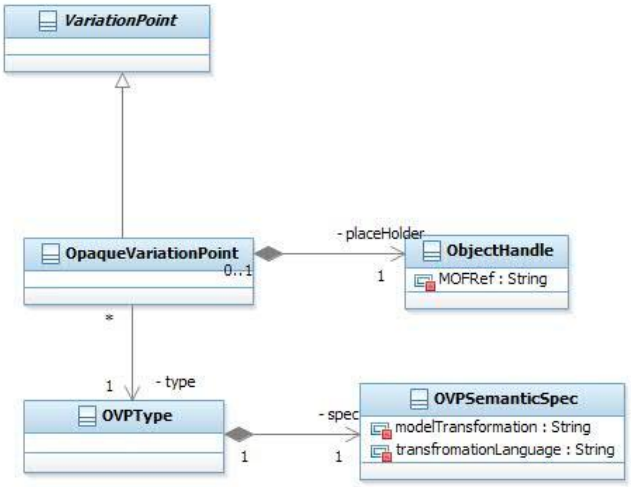

Figure 7 Opaque Variation Point

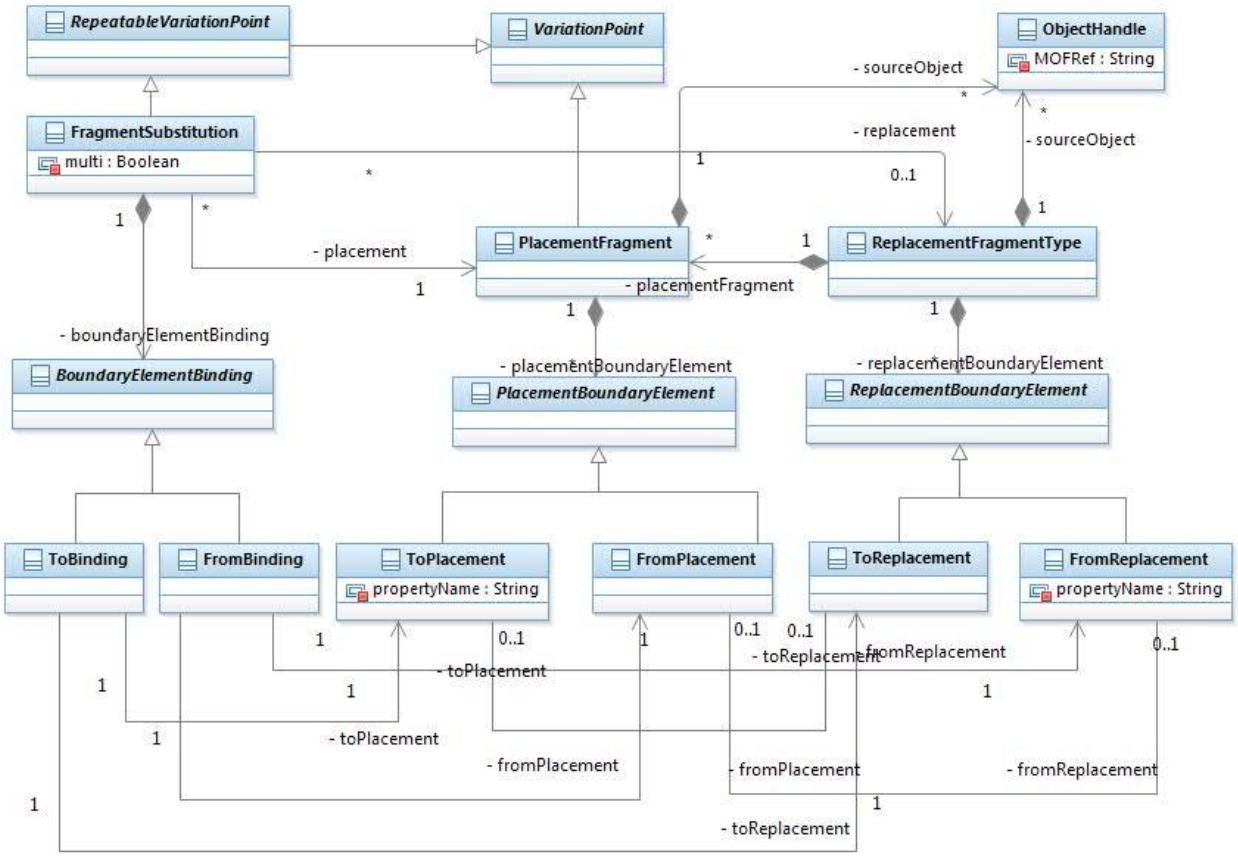

Figure 8 Fragment Substitution

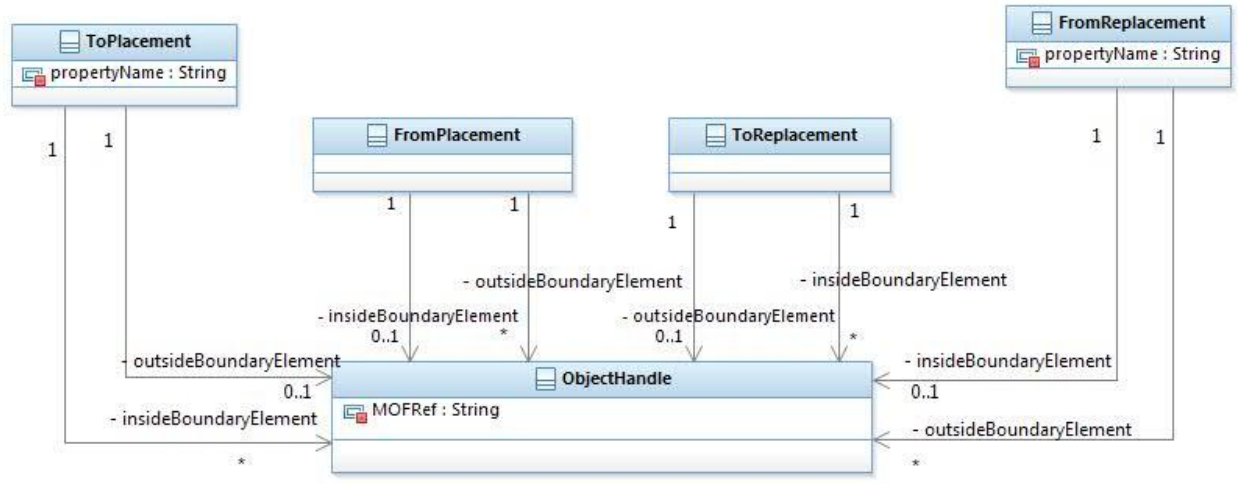

Figure 9 Fragment Substitution Boundary Points

3 BVR Metamodel Manual

3.1 BCLConstraint

3.1.1 Description

Top class of a constraint. Contains basic constraint language expressions.

3.1.2 Generalizations

Constraint

3.1.3 Associations

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

3.2 BCLExpression

3.2.1 Description

A generic class for expressions of the basic constraint language (BCL).

3.3 BooleanLiteralExp

3.3.1 Description

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

3.3.2 Generalizations

BCLExpression

3.3.3 Attributes

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

3.4 BoundaryElementBinding

3.4.1 Description

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

3.4.2 Generalizations

NamedElement

3.5 BVRModel

3.5.1 Description

BVRModel holds the whole BVR model with abstraction layer, realization layer and base models.

3.5.2 Generalizations

VPackage

3.5.3 Associations

- baseModelBase : ObjectHandle [1..1]
  The base model which will be used as starting point for creating products in the base language
- baseModelLibraries : ObjectHandle [0..*]
  Base libraries used additively in creating products from the baseModelBase
- realizationModel : VariationPoint [0..*]
  The realizationModel consists of the variation points used to respresent the realization of the variabilityModel given the resolutionModel. The variation points define the changes done with the base model to fulfill the configurations specified.
- resolutionModels : CompoundResolution [0..*]
  Resolution models are defining the configurations of the products
- variabilityModel : CompoundNode [1..1]
  The variability model (VSpec tree) corresponding to feature models

3.6 Choice

3.6.1 Description

A Choice is a VSpec and a VNode that represents a yes/no decision. When a VariationPoint is bound to a choice it is dependent upon whether the resolution is a PosResolution or a NegResolution to determine what VariationPoint to execute.

3.6.2 Generalizations

CompoundNode
VSpec

3.6.3 Attributes

- defaultResolution : Boolean [0..1]
  The default resolution of this choice. (If True then the default is a PosResolution, if False then the default is a NegResolution)
- 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.

For a root choice, True implies it must be resolved positively.

3.6.4 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))

3.7 ChoiceOccurrence

3.7.1 Description

ChoiceOccurrence is similar to a Choice, but refers to a VType instead of having the ability to define a subordinate VNode structure.

3.7.2 Generalizations

VNode
VSpec

3.7.3 Associations

- vType : VType [1..1]
  vType refers to the VType that defines the details of the ChoiceOccurrence

3.8 ChoiceResolution

3.8.1 Description

A ChoiceResolution is a VSpecResolution which resolves a single choice positively (PosResolution) or negatively (NegResolution).

3.8.2 GENERALIZATIONS

VSpecResolution

3.8.3 ASSOCIATIONS

- resolvedChoice : Choice [1..1]
  The resolved choice of this resolution.
- resolvedChoiceOcc : ChoiceOccurrence [1..1]
  The resolved choice occurrence
- resolvedVClassifier : VClassifier [1..1]
  The resolved classifier
- resolvedVClassOcc : VClassOccurrence [1..1]
  The resolved VClass occurrence

3.8.4 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.

3.9 ChoiceVariationPoint

3.9.1 Description

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

3.9.2 Generalizations

VariationPoint

3.9.3 Attributes

- resolution_kind : Boolean [1..1]
  The resolution_kind has the following definition:True: The enclosing ChoiceVariationPoint shall be executed if the triggering resolution is a PosResolution.False: The enclosing ChoiceVariationPoint shall be executed if the triggering resolution is a NegResolution.

3.9.4 Associations

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

3.10 CompoundNode

3.10.1 Description

CompoundNodes are roots in VNode subtrees.

3.10.2 Generalizations

VNode

3.10.3 Associations

- member : VNode [0..*]
  The members of the compound node constituting the next level of the VSpec tree.
- target : Target [0..*]
  Targets owned by this CompoundNode.

3.11 CompoundResolution

3.11.1 Description

CompoundResolutions define roots of subtrees of VSpecResolutions.

3.11.2 Generalizations

ChoiceResolution

3.11.3 Associations

- members : VSpecResolution [0..*]
  The members of the compound resolution constituting the next level of the resolution tree.

3.12 Constraint

3.12.1 Description

A constraint specifies restrictions on permissible resolution models. A constraint can refer to any entity in the closest namespace in which it is contained. Entities in other namespaces may be referred through remote notation. Any entity referred by a constraint must be uniquely determined.

3.12.2 Generalizations

NamedElement

3.13 FragmentSubstitution

3.13.1 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:
    a. FromBinding: fromReplacement.insideBoundaryElement.propertyName[] = fromPlacement.outsideBoundaryElement[]
    b. 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.

3.13.2 GENERALIZATIONS

ChoiceVariationPoint
RepeatableVariationPoint

3.13.3 ATTRIBUTES

- multi : Boolean [1..1]
  Indicates multiple fragment substitution meaning that the substitution can be executed several times without removing the placement more than once.

3.13.4 ASSOCIATIONS

- boundaryElementBinding : BoundaryElementBinding [0..*]
  Specifies the binding between the placement and replacement fragments.
- placement : PlacementFragment [1..1]
  Specifies the fragment to be replaced. The placement fragment will be removed once.
- replacement : ReplacementFragmentType [0..1]
  Specifies the replacement fragment that will replace the placement.

3.13.5 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.

Note : FragmentSubstitution needs the use of FromBinding and ToBinding metaclasses (please refer to the corresponding pre and post conditions).

3.14 FromBinding

3.14.1 Description

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

3.14.2 Generalizations

BoundaryElementBinding

3.14.3 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.

3.14.4 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)))

3.15 FROMPLACEMENT

3.15.1 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.

3.15.2 GENERALIZATIONS

PlacementBoundaryElement

3.15.3 ASSOCIATIONS

- fromReplacement : FromReplacement [0..1]
  Reference to a FromReplacement in a containing replacement fragment.
- insideBoundaryElement : ObjectHandle [0..1]
  Insiude boundary elements refer to elements inside the fragment.
- outsideBoundaryElement : ObjectHandle [0..*]
  Outside Model Elements are element that are referred by the model elements inside the placement fragment (but which are themselves not inside the placement fragment).

3.16 FROMREPLACEMENT

3.16.1 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.

3.16.2 GENERALIZATIONS

ReplacementBoundaryElement

3.16.3 ATTRIBUTES

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

3.16.4 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 distinguish multiplicity references.

3.17 IntegerLiteralExp

3.17.1 Description

A literal expression that represents integer numbers.

3.17.2 Generalizations

NumericLiteralExp

3.17.3 Attributes

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

3.18 MultiplicityInterval

3.18.1 Description

A MultiplicityInterval specifies lower and upper multiplicities.

3.18.2 Attributes

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

3.18.3 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)

3.19 NamedElement

3.19.1 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.

3.19.2 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.

3.19.3 Associations

- note : Note [0..*]
  The note is some additional information about this element. It can be anything and it will not be interpreted as significant by the BVR language, but different tooling can interpret the information ad lib.

3.20 NegResolution

3.20.1 Description

Negative Resolution. This means that the decision is negative and there is no need to continue down the eventual VNode tree referred by this NegResolution.

3.20.2 Generalizations

ChoiceResolution

3.21 Note

3.21.1 Description

Notes are auxiliary information associated with the NamedElements, typically associated with VSpecs and the likes. When the kind is empty it is just a text.

3.21.2 GENERALIZATIONS

NamedElement

3.21.3 ATTRIBUTES

- expr : String [1..1]
  expr is the content of the Note. The interpretation of expr is given by the kind. It is tool-specific whether and how the Notes are processed by the tool.
- kind : String [1..1]
  kind is a String that designates what this Note is about. kind can typically be an indication about some extra-functional property such as "response time", or "cost". When kind is empty this means that the Note is a plain comment.

3.22 NUMERICLITERALEXP

3.22.1 DESCRIPTION

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

3.22.2 GENERALIZATIONS

BCLExpression

3.23 OBJECTHANDLE

3.23.1 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.

3.23.2 ATTRIBUTES

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

3.24 OBJECTSPECIFICATION

3.24.1 DESCRIPTION

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

3.24.2 GENERALIZATIONS

ValueSpecification

3.24.3 Associations

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

3.25 ObjectType

3.25.1 Description

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

3.25.2 Generalizations

Variabletype

3.25.3 Attributes

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

3.26 OpaqueConstraint

3.26.1 Description

An Opaque Constraint imposes additional restrictions that cannot be expressed in the basic constraint language. Opaque constraints can be expressed in any language (given by the language attribute) but cannot be directly understood as BVR. Typically an opaque constraint must be used if there is a need to express universal quantification or other advanced expressiveness. OCL is an obvious choice for such a more advanced constraint language.

3.26.2 Generalizations

Constraint

3.26.3 Attributes

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

3.27 OpaqueVariationPoint

3.27.1 Description

An OpaqueVariationPoint is an executable, domain-specific variation point whose semantics is not defined by BVR. 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.

3.27.2 Generalizations

VariationPoint
VariationPoint

3.27.3 Associations

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

3.28 <Enumeration>Operation

3.28.1 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)

3.28.2 Literals

- arithDiv
  Arithmetic division
- arithMinus
  Arithmetic subtraction
- arithMult
  Arithmetic multiplication
- arithNeg
  Arithmetic negation

- arithPlus
  Arithmetic addition
- eq
  Equal
- gt
  Greater than
- gte
  Greater than or equal
- isDefined
  Checks if the value is defined.
- isUndefined
  Check if the value is undefined.
- logAnd
  Logical and
- logIff
  Logical if and only if
- logImplies
  Logical implies
- logNot
  Logical not
- logOr
  Logical or
- logXor
  Logical xor
- lt
  Logical less than
- lte
  Logical less than or equal
- strConc
  String concatenation

3.29 OperationCallExp

3.29.1 Description

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

3.29.2 Generalizations

BCLExpression

3.29.3 Associations

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

3.30 OVPSemanticSpec

3.30.1 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.

3.30.2 Attributes

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

3.31 OVPType

3.31.1 Description

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

3.31.2 Generalizations

VPackageable

3.31.3 Associations

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

3.32 ParametricSlotAssignment

3.32.1 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 slotIdentifier. 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

3.32.2 Generalizations

ParametricVariationPoint

3.32.3 Attributes

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

3.32.4 Associations

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

3.32.5 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 ParametricSlotAssignment::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)

3.33 ParametricVariationPoint

3.33.1 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.

3.33.2 Generalizations

VariationPoint

3.33.3 Associations

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

3.34 PlacementBoundaryElement

3.34.1 Description

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

3.34.2 Generalizations

NamedElement

3.35 PlacementFragment

3.35.1 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.

3.35.2 Generalizations

VariationPoint

3.35.3 Associations

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

- sourceObject : ObjectHandle [0..*]

3.36 PosResolution

3.36.1 Description

Positive Resolution. This means that one needs to resolve the eventual CompoundNode referred by this PosResolution.

3.36.2 Generalizations

CompoundResolution

3.37 <Enumeration>PrimitiveTypeEnum

3.37.1 Description

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

3.37.2 Literals

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

3.38 PrimitiveValueSpecification

3.38.1 Description

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

3.38.2 Generalizations

ValueSpecification

3.38.3 Associations

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

3.39 PrimitveType

3.39.1 Description

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

3.39.2 Generalizations

Variabletype

3.39.3 Attributes

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

3.40 RealLiteralExp

3.40.1 Description

A literal expression that represents floating-point numbers.

3.40.2 Generalizations

NumericLiteralExp

3.40.3 Attributes

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

3.41 RepeatableVariationPoint

3.41.1 Description

A repeatable variation point is a variation point that may be applied several times during materialization. It may only be bound to a VClassifier (or VClassOccurrence) and is applied once for each resolution referring to it.

3.41.2 Generalizations

VariationPoint

3.41.3 Associations

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

3.42 ReplacementBoundaryElement

3.42.1 Description

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

3.42.2 Generalizations

NamedElement

3.43 ReplacementFragmentSpecification

3.43.1 Description

A value of ReplacementFragmentType

3.43.2 Generalizations

ValueSpecification

3.43.3 Associations

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

3.44 ReplacementFragmentType

3.44.1 Description

Replacement Fragment Type defines a fragment of the base model that will be used as replacement for some 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.

3.44.2 Generalizations

Variabletype

3.44.3 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.
- sourceObject : ObjectHandle [0..*]

3.45 ResolutionLiteralDefinition

3.45.1 Description

Defines Resolution Literal as a named subtree of VSpecResolutions. This can then be reused through ResolutionLiteralUses. Typically ResolutionLiteralDefinitions are used to name and store subproducts that are either existing or planned.

3.45.2 Generalizations

CompoundResolution

3.45.3 Associations

- literalType : VType [0..1]
  The VType defining the details of the ResolutionLiteralDefinition. There must be pairwise correspondence between elements of the ResolutionLiteralDefinition tree and that of the literalType tree.

3.45.4 Semantics

Invariant :
Let the ResolutionLiteralDefinition be 'rlit' and its owner 'rlitown' then rlitown.resolvedVSpec owns rlit.literalType.
Informally this means that the resolution literals are placed in the resolution model parallel to where the corresponding VType is placed in the variability model.

3.46 ResolutionLiteralUse

3.46.1 Description

The use of a resolution literal. This is like an instance of a resolution type.

3.46.2 Generalizations

ChoiceResolution

3.46.3 Associations

- defLiteral : ResolutionLiteralDefinition [0..1]
  Referring to the resolution literal definition of this use

3.47 SlotAssignment

3.47.1 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.

3.47.2 Generalizations

ChoiceVariationPoint

3.47.3 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.

3.47.4 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.

3.47.5 Semantics

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

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.

3.48 StagedVariationPoint

3.48.1 Description

A staged variation point refers to occurrences of VTypes (either VClassOccurrence or ChoiceOccurrence). The point is that realizing occurrences may vary from occurrence to occurrence of the very same VType. Typically the different occurrences of a VType correspond to desired changes in different concrete places of the base model.

3.48.2 Generalizations

VariationPoint

3.48.3 Associations

- bindingStagedVP : VSpec [1..1]
  The binding CVSpec which refers either a ChoiceOccurrence or a VClassOccurrence.
- member : VariationPoint [0..*]
  Members of the staged variation point constituting the next level of the tree and corresponding pairwise with the members of the VType of the occurrence referred by the bindingStagedVP

3.49 StringLiteralExp

3.49.1 Description

A literal expression that represents strings.

3.49.2 Generalizations

BCLExpression

3.49.3 Attributes

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

3.50 Target

3.50.1 Description

Target represents the substance of what a VSpec is about. VSpecs are in fact variability decision points, and even in the same VNode tree there may be several VSpecs that refer to the same substance, but the paths to the decision point differ. Constraints talk about Targets. The substance of what a VSpec is about does not necessarily mean a piece of the base model, but may sometimes refer to exactly that. The essence of "substance" is that the same abstract decision can be reached in different ways in the VSpec tree (variability model).

3.50.2 Generalizations

NamedElement

3.51 TargetRef

3.51.1 Description

A constraint expression that references a Target and through that a set of VSpecs. It must reference existing VSpecs.

3.51.2 Generalizations

BCLExpression

3.51.3 Associations

- target : Target [1..1]
  Reference to the Target.

3.52 ToBinding

3.52.1 Description

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

3.52.2 Generalizations

BoundaryElementBinding

3.52.3 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.

3.52.4 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)))

3.53 ToPlacement

3.53.1 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

3.53.2 Generalizations

PlacementBoundaryElement

3.53.3 Attributes

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

3.53.4 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.

3.53.5 SEMANTICS

Invariant :
Constraint :
1) self.outsideBoudaryElement <> null xor outsideRef != null
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)))

3.54 TOREPLACEMENT

3.54.1 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.

3.54.2 GENERALIZATIONS

ReplacementBoundaryElement

3.54.3 ASSOCIATIONS

- insideBoundaryElement : ObjectHandle [0..*]
  Model elements that are referred to by outside model elements.
- outsideBoundaryElement : ObjectHandle [0..1]

- toPlacement : ToPlacement [0..1]
  Reference to a ToPlacement contained by the replacement fragment.

3.55 UnlimitedLiteralExp

3.55.1 Description

A literal expression that represents unlimited natural numbers.

3.55.2 Generalizations

NumericLiteralExp

3.55.3 Attributes

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

3.56 ValueResolution

3.56.1 Description

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

3.56.2 Generalizations

VSpecResolution

3.56.3 Associations

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

3.56.4 Semantics

Dynamic semantics
Pre and post condition
Pre-condition :
Before the execution of a ValueResolution, the variable stored in the ValueResolution must exist in the variable dictionary.
Post-condition:
After a ValueResolution has been executed, the resolvedVariable is assigned with the value.

3.57 ValueSpecification

3.57.1 Description

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

3.57.2 Associations

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

3.58 Variable

3.58.1 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.

3.58.2 Generalizations

VSpec

3.58.3 Associations

- defaulValue : ValueSpecification [0..1]
  The default value of this Variable.
- type : Variabletype [1..1]
  The type of the variable.

3.59 Variabletype

3.59.1 Description

The type of a variable or a value specification.

3.59.2 Generalizations

VPackageable

3.60 VariationPoint

3.60.1 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.

3.60.2 Generalizations

NamedElement

3.60.3 Associations

- bindingVSpec : VSpec [1..1]
  The VSpecs to which the variation point is bound.
- stagedVariationPoint : StagedVariationPoint [0..1]

3.61 VClassifier

3.61.1 Description

VClassifier is a set concept. The instanceMultiplicity defines the range of how many ChoiceResolutions that can be referring this particular VClassifier. A VClassifier can also have a subtree as it is a CompoundNode which defines subordinate variability specifications (VNodes). When a repeatable variation point is bound to a VClassifier it will be applied once for each resolution of the VClassifier during materialization.

3.61.2 Generalizations

CompoundNode
VSpec

3.61.3 Associations

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

3.62 VClassOccurrence

3.62.1 Description

VClassOccurrence is a set concept similar to VClassifier. The difference is that VClassOccurence refers a VType and does not have its own defining subtree.

3.62.2 Generalizations

VNode
VSpec

3.62.3 Associations

- instanceMultiplicity : MultiplicityInterval [0..*]
- vType : VType [1..1]

3.63 VNode

3.63.1 Description

VNodes define the trees structures representing implicit logical constraints on the resolution of the children. Choice, ChoiceOccurrence, VClassifier and VClassOccurrence are all both VNode and VSpec, while a Variable is a VSpec, but not a VNode. VNodes own Variables, however. A VNode may optionally have a group multiplicity specifying upper and lower multiplicities against its children. The meaning of this is that each positive resolution (PosResolution) against a VSpec which is also a VNode must have a number of positive child resolutions conforming to the multiplicity interval.

3.63.2 Associations

- groupMultiplicity : MultiplicityInterval [0..1]
  The group multiplicity of the VNode. If the corresponding VSpec is resolved positively and has a group multiplicity then the number of
its children resolved positively must conform to the specified multiplicity interval.
- ownedConstraint : Constraint [0..*]
  Constraints over the VSPecs over this VInterface
- variable : Variable [0..*]
  Owned variables.

3.64 VPackage

3.64.1 Description

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

3.64.2 Generalizations

VPackageable

3.64.3 Associations

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

3.65 VPackageable

3.65.1 Description

A VPackageable is an element that may be owned by a package. VPackageable also represent namespaces. A namespace is where the names of the named elements must be unique. Names of other namespaces may be reached by remote notation.

3.65.2 Generalizations

NamedElement

3.66 VRef

3.66.1 Description

Variability Reference. A VRef is a reference to a node in the VSpec tree (VNode).

3.66.2 Generalizations

Variabletype

3.66.3 Associations

- refVSpec : VSpec [0..*]
  The VSpec referred by the VRef

3.67 VRefValueSpecification

3.67.1 Description

VRefValueSpecification defines the value of a VRef. The following invariant must hold: VRefValueSpecification.vSpecResolution.resolvedVSpec == VRefValueSpecification.type.refVSpec (with the appropriate qualifiers when there is multiplicity)

3.67.2 Generalizations

ValueSpecification

3.67.3 Associations

- type : VRef [1..1]

- vSpecResolution : VSpecResolution [0..*]
  The resolution reference such that VSpecResolution.resolvedVSpec == type.refVSpec

3.68 VSpec

3.68.1 Description

VSpec is the metaclass for Variability Specifications. VSpecs are the decision points that need to be resolved, and therefore the VResolutionSpecifications refer to VSpecs. Furthermore, the VSpecs are what controls the materialization and therefore VSpecs are referred by VariationPoints. There are more specifics relating to the specializations of VSpec, VResolutionSpecification and VariationPoint

3.68.2 Generalizations

NamedElement

3.68.3 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 current BVR.

3.68.4 Associations

- target : Target [1..1]
  The Target defining the substance of the choice represented by this VSpec. (Many VSpecs may refer the same Target meaning that all these VSpec decision points refer to the same substance)

3.68.5 Semantics

Invariant :
If the VSpec has a multiplicity interval, it must have a number of children included between the lowerMultiplicity and the upperMultiplicity

3.69 VSpecResolution

3.69.1 Description

A VSpecResolution resolves a VSpec. VSpecResolutions are organized as trees, mirroring partially the tree structure of the VSpecs they resolve.

3.69.2 Generalizations

NamedElement

3.69.3 Associations

- resolvedVSpec : VSpec [1..1]
  The VSpec this VSpecResolution resolves.Due to VSpec inheritance and VClassifiers, a given

VSpec may have several VSpecResolutions resolving it, where each resolution is in the context of its parent.

3.69.4 SEMANTICS

Invariant :
We must retrieve at least all of the resolvedVSpec's children associated with the VSpecResolution's vspec children

OCL :
-- VSpecResolution
-- We must retrieve at least all of the resolvedVSpec's children associated with the VSpecResolution's vspec children
-- So, the number of VSpecResolution's children must be >= to the number of resolvedVSpec' s children
context VSpecResolution
inv VSpecResChildrenCorrespondsToVSpecChildren :
self.childResolution->size() >= self.resolvedVSpec.childVSpec->size()
and
not (self.resolvedVSpec.childVSpec->exists (vSpec | not (self.childResolution->exists(vRes | vRes.resolvedVSpec == (VSpec)))))

3.70 VTYPE

3.70.1 DESCRIPTION

VType is a type of VNodes which means that it represents a pattern of reuse - a subtree of a VNode tree. VTypes are referred by ChoiceOccurrences and VClassOccurrences. Thus we may informally understand a VType by the following formulas: ChoiceOccurrence+VType == Choice ; VClassOccurrence+VType == VClassifier.

3.70.2 GENERALIZATIONS

CompoundNode
VPackageable

4 Abbreviations and Definitions

Term	Definition
BVR	Base Variability Resolution models
CVL	Common Variability Language

5 REFERENCES

Author, Year — Authors; *Title*; Publication data (document reference)

VARIES-TA — ARTEMIS Call 2011, ARTEMIS-2011-1, 295397 VARIES: VARiability In safety-critical Embedded Systems. Technical Annex.

Appendix I. PARTNER IDENTIFICATION

Short name	Full name	Type	Country
Atego DE	Atego GmbH	SME	DE
Atego UK	Atego Systems Ltd	SME	UK
Autronica	Autronica Fire & Security AS	LE	NO
Barco	Barco NV	LE	BE
B&M	Berner & Mattner Systemtechnik GmbH	LE	DE
FMTC	Flanders' Mechatronics Technology Centre vzw	RES	BE
Fraunhofer	Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V.	RES	DE
HI-Iberia	HI-Iberia Ingenieria y Proyectos SL	SME	ES
HiQ	HiQ Finland Oy	LE	FI
IMDEA	Fundation IMDEA Software	RES	ES
ITU	IT University of Copenhagen	RES	DK
Macq	Macq	SME	BE
Metso	Metso Automation Oy	LE	FI
Mobisoft	Mobisoft Oy	SME	FI
pure-systems	pure-systems GmbH	SME	DE
SINTEF	STIFTELSEN SINTEF	RES	NO
Sirris	Sirris c.d.g.	RES	BE
SKS	SoftKinetic Sensors	SME	BE
Spicer	Spicer Off-Highway NV (Dana)	LE	BE
TECNALIA	TECNALIA Corporación Tecnológica	RES	ES
TÜV	TÜV SÜD AG	LE	DE
Vlerick	Vlerick Business School	RES	BE
VTT	VTT technical research centre of Finland	RES	FI

Appendix II. BVR – BETTER VARIABILITY RESULTS

Øystein Haugen[1] and Ommund Øgård[2]

[1]SINTEF
PO Box 124 Blindern
NO-0314 Oslo, Norway
oystein.haugen@sintef.no

[2]Autronica Fire & Security
Postboks 5620.
7483 Trondheim,
Ommund.Ogaard@autronicafire.no

Abstract. We present BVR (Base Variability Resolution models), a language developed to fulfill the industrial needs in the safety domain for variability modeling. We show how the industrial needs are in fact quite general and that general mechanisms can be used to satisfy them. BVR is built on the OMG Revised Submission of CVL (Common Variability Language), but is simplified and enhanced relative to that language.

II.1 INTRODUCTION

BVR (Base Variability Resolution models) is a language built on the Common Variability Language [1-3] technology, but enhanced due to needs of the industrial partners of the VARIES project[2], in particular Autronica. BVR is built on CVL, but CVL is not a subset of BVR. In BVR we have removed some of the mechanisms of CVL that we are not using in our industrial demo cases that apply BVR. We have also made improvements to what CVL had originally.

Our motivation has mainly been the Fire Detection demo case at Autronica, but we have also been inspired by the needs of the other industrial partners of VARIES through their expressed requirements to a variability language.

This paper contains a quick presentation of the Common Variability Language in Chapter II.2. In Chapter II.3 we relate our work to its motivation in the Autronica fire alarm systems, but argue that we need a more compact and pedagogical example and our car case is presented in Chapter II.4. Then we walk through our new BVR concepts in Chapter II.5 and discuss the suggested improvements in Chapter II.6 and conclude in Chapter II.7.

II.2 CVL – THE COMMON VARIABILITY LANGUAGE

The Common Variability Language is the language that is now a Revised Submission in the OMG [3] defining variability modeling and the means to generate product models. CVL is in the tradition of modeling variability as an orthogonal, separate model such as OVM [4] and the MoSiS CVL [1] which formed one of the starting points of the OMG CVL. The principles of separate variability model and how to generate product models are depicted in Fig. 10

[2] http://www.varies.eu

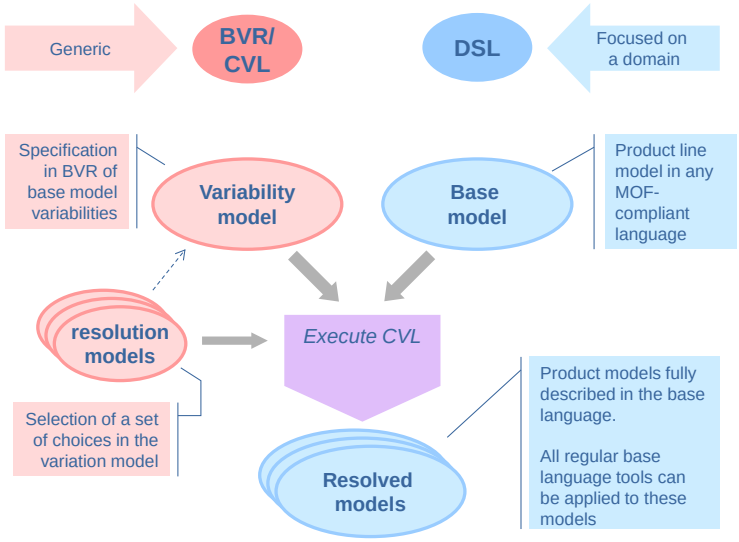

Fig. 10. CVL principles

The CVL architecture is described in Fig. 11. It consists of different inter-related models. The variability abstraction consists of a VSpec model supplemented with constraints, and a corresponding resolution model defining the product selections.

The variability realization contains the variation points representing the mapping between the variability abstraction and the base model such that the selected products can be automatically generated. The configurable units define a layer intended for module structuring and exchange. In this paper we have not gone into that layer.

The VSpec model is an evolution of the FODA [5] feature models, but the main purpose of CVL has been to provide a complete definition such that product models can be generated automatically from the VSpec model, the resolution model and the realization model.

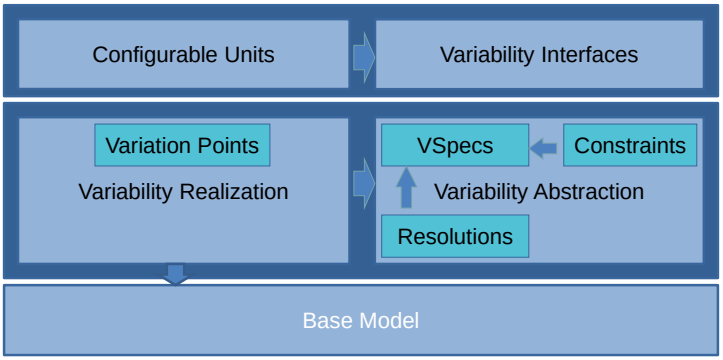

Fig. 11. CVL architecture

BVR (named from Base, Variability, Resolution models) is an evolution from CVL where some constructs have been removed for improved simplicity and some new constructs have been added for better and more suited expressiveness. The new constructs are those presented in this paper.

II.3 The Autronica Fire Detection Case

The main motivator has been the Autronica Fire Detection Case. Autronica Fire & Security[3] is a company based in Trondheim that delivers fire security systems to a wide range of high-end locations such as oil rigs and cruise ships. Their turnover is around 100 MEUR a year.

The Autronica demo case is described schematically in Fig. 12.

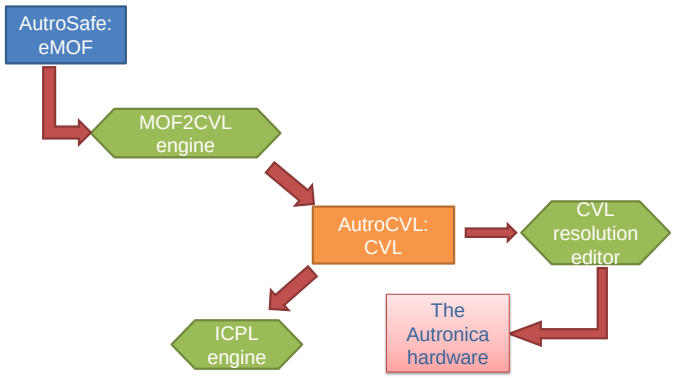

Fig. 12. The Autronica demo case

The purpose of the demo case was to explore the ways in which the Autronica specific model "AutroSafe" could be applied for two different purposes. Firstly, after transforming the MOF metamodel into CVL the CVL tools could be used to define AutroSafe configurations. Secondly, and possibly more interestingly, from the CVL description it would be possible to apply analysis tools to AutroSafe which were made generally for CVL. In particular the ICPL tool [6-8] could be used to find an optimal set of configurations to test AutroSafe.

For our purpose in this paper, the Autronica use-case provided the real background for understanding what kind of product line they have to manage. In performing our use case at Autronica we explored the transition from the AutroSafe model to a CVL model. The AutroSafe model was a UML model that can be understood as a reference model or a conceptual model of how the fire detection system concepts are associated [9]. We realized that this conceptual model could be considered a metamodel which could be used to generate language specific editors that would be limited to describing correct fire detection systems. Furthermore, we realized that the conceptual model could be used as base for a transformation leading to a variability model. We explored this route by manually transforming the AutroSafe metamodel through transformation patterns that we invented through the work. At the same time Autronica explored defining variability models for parts of the domain directly, also for the purpose of using the variability model to generate useful test configurations.

II.4 The Example Case – The Car Configurator

Since the Autronica case is rather large and requires special domain knowledge we will illustrate our points with an example case in a domain which most people can relate to, namely to configure the features of a car.

Our example case is that of configuring a car. In fact our starting point for making the variability model was the online configurator for Skoda Yeti in Norway[4], but we have made some adaptations to suit our purpose as example.

[3] http://www.autronicafire.com

Our car product line consists of diesel cars that can have either manual or automatic shift. The cars with automatic shift would only be with all wheel drive (AWD) and they would need the 140 hp engine. On the other hand the cars with manual shift had a choice between all-wheel drive and front drive. The front wheel drive cars were only delivered with the weaker 110 hp engine, while the all-wheel drive cars had a choice between the weak (110 hp) or the strong (140 hp) engine.

Following closely the natural language description given above we reach the CVL model shown in Fig. 13.

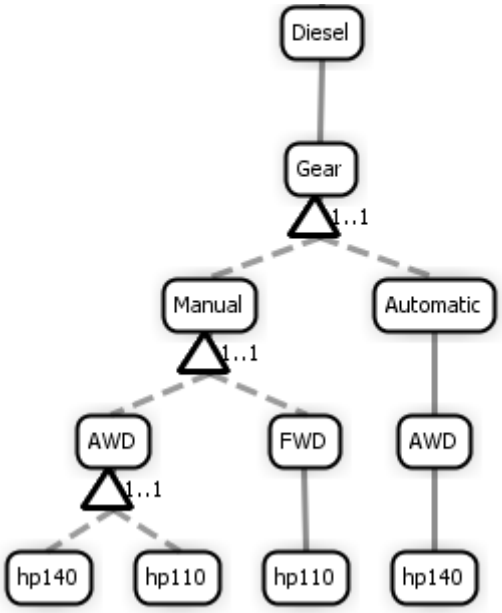

Fig. 13. The example diesel car in CVL

For readers unfamiliar with CVL they should appreciate that solid lines indicate that the child feature (or VSpecs as we call them in BVR/CVL) is mandatory when the parent is present in a resolution. Dashed lines on the other hand indicate optionality. A small triangle with associated numbers depicts group multiplicity giving the range of how many child VSpecs must and can be chosen. Thus when *AWD* has children *hp140* and *hp110* associated with a group multiplicity of 1..1, this means that if *AWD* is chosen, at least 1 and at most 1 out of *hp140* and *hp110* must be selected.

II.5 The BVR Enhancements

In this chapter we will walk through the enhancements that we have made to accommodate for general needs inspired by and motivated by industrial cases.

II.5.1 Targets – the power of the variability model tree structure

Our CVL diagram in Fig. 13 is not difficult to understand even without the natural language explanation preceding it given some very rudimentary introduction to CVL diagrams (or feature models for that matter). We see that the restrictions are transparently described through the tree structure and our decisions are most easily done by traversing the tree from the top.

It is also very obvious that the diesel car has only one engine, and that it has only one gear shift and one kind of transmission. Therefore everybody understands that even though there are two elements

[4] http://cc-cloud.skoda-auto.com/nor/nor/nb-no/

named "hp140" they refer to the same target, namely the (potential) strong engine. In the same way "AWD" appears twice in the diagram, but again they both refer to the same target. It turns out that CVL and other similar notations do not clearly define this. In fact CVL defines that the two choices named "hp140" are two distinct choices with no obvious relationship at all.

When does this become significant? Does it matter whether the two choices refer to the same target? It turns out that it does both for conceptual reasons and technical ones.

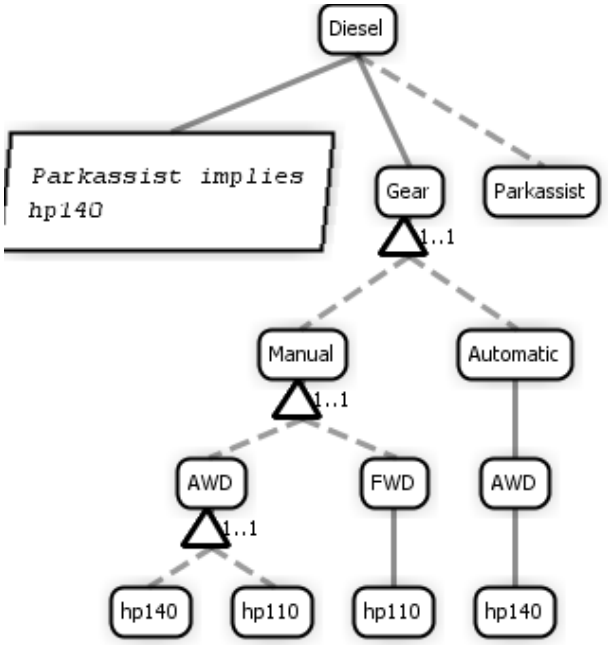

Fig. 14. Adding a Parking assistant

In Fig. 14 we have added an optional parking assistant to the car. However, to be allowed a parking assistant, you need to buy the strong engine. This is intuitive and easily understood, but formally this means that any of the occurrences of "hp140" should satisfy the constraint. Thus, constraints talk about the targets and not the choices.

II.5.2 Beyond one tree

We see that the tree structure of variability models convey in a very transparent way the restrictions of the decisions to be made. However, trees are sometimes not enough. In our Autronica experiment we wanted to reflect in the CVL model the structure of variability in a way that would abstract the actual configurations of fire detection systems in airports and cruise ships. In this way our variability model became close to the structures of the base model. Our car example model has the opposite focus as it highlights the restrictions of interrelated decisions.

In variability models that are close to the base model one can expect that tree structures are insufficient to describe the necessary relationships and in the Autronica case the physical layout of detectors and alarms was overlaid by an equally important structure of logical relationships and groups. To represent the alternative, overlaid structures we need ways to refer between variability elements and our obvious suggestion is to introduce references (or pointers as they are also called).

References can also serve as traces and indicate significant places in other parts of the model.

In our Autronica experiment we had to encode references since references were not available as a concept in CVL. By encoding references we used integers to indicated identifiers and corresponding

pointers. This required a lot of manual bookkeeping which turned out to be virtually impossible to perform and even more impossible to read.

In BVR we want to reflect the physical structure that is represented in the conceptual model as composition through the main hierarchical VSpec tree. The logical structure that is modeled by associations in the conceptual UML model would be represented by variability references in BVR.

II.5.3 From proper trees to properties

Judging from the tool requirements elicited from the VARIES partners, they wanted a lot of different information stored in the variability (and resolution) models. Some of the information would be intended for their own proprietary analysis tools, and sometimes they wanted to associate temporary data in the model.

When working with the Autronica case and experiencing the difficulties with encoding the needed references we ourselves found that we wished that we had a way to explain the coding in a natural language sentence. Thus we felt the very common need for having comments.

II.5.4 Reuse and Type – the first needs for abstraction

Fire alarming is not trivial. Autronica delivers systems with thousands of detectors and multiple zones with or without redundancy to cruise ships and oil rigs where running away from the fire location altogether is not the obvious best option since the fire location is not easily vacated. In such complicated systems it was not a big surprise that recurring patterns would be found.

Without going into domain-specific details, an AutroSafe system will contain IO Modules. Such IO modules come in many different forms and they represent a whole product line in its own right, this actually applies for most of the parts a fire alarm system is composed of, e.g., smoke detectors, gas detectors, panels etc. Some IO modules may be external units and such external units may appear in several different contexts. As can be guessed, external units have a very substantial variability model and it grows as new detectors come on the market.

In our experiment we encoded these recurring patterns also by integers as we did with references with the same plethora of integers and need for bookkeeping as a result. It was clear that concepts for recurring patterns would be useful in the language. We investigate introducing a type concept combined with occurrences referring the types.

Our example car product line has no complicated subproduct line, but we have already pointed out that AWD recurs twice in the original model. We express AWD as a type and apply two occurrences of it.

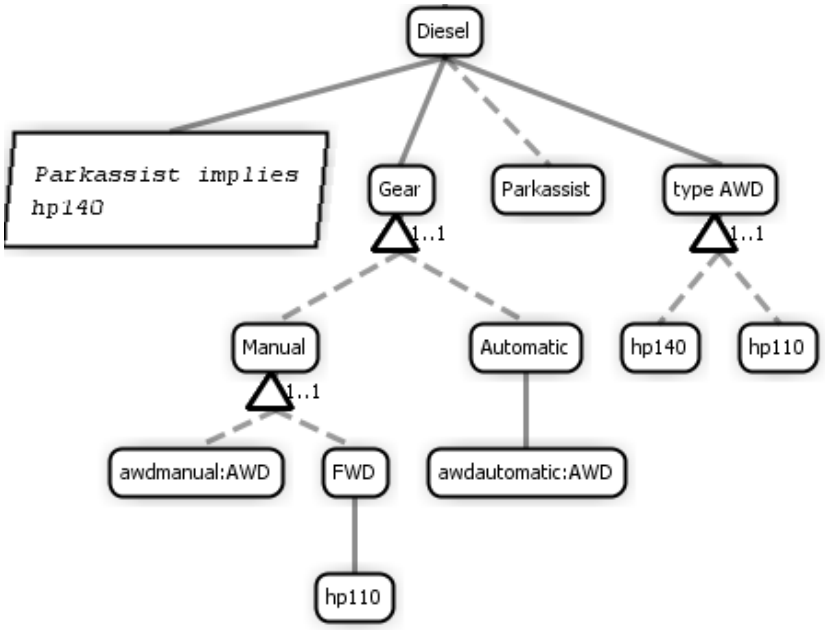

Fig. 15. The AWD variability type

The observant reader will have seen that replacing the two occurrences of *AWD* in Fig. 15 with replications of the type will not yield exactly the tree shown in Fig. 14 since for Automatic shift only the strong engine can be chosen. Such specialization should be expressed by a constraint associated with the occurrence.

We note that the type itself is defined on a level in the tree which encloses all of the occurrences. It is indeed not obvious where and how the type should be defined and we have shown here what was sufficient to cover the Autronica case.

In Fig. 15 *awdautomatic:AWD* is a ChoiceOccurrence which represents an occurrence or instantiation of the *AWD* VType. A question is whether a ChoiceOccurrence can itself contain a tree structure below it since it is indeed VNode? If there was a subtree with a ChoiceOccurrence as root, what would be the semantics of that tree acknowledging that the referred VType defines a tree, too? It is quite obvious that there must be some consistency between the occurrence tree and the corresponding VType tree. Intuitively the occurrence tree should define a narrowing of the VType tree. There are, however, some serious challenges with this. Firstly, to specify the narrowing rules syntactically is not trivial. Secondly, to assert that the narrowing rules are satisfied may not be tractable by the language tool. Thirdly, the narrowing structures may not be intuitive to the user. Therefore, we have decided that only constraints will be allowed to be used to further specify a choice occurrence. In our example case, the diagram in Fig. 15 would add a constraint below *awdautomatic:AWD* with exactly one target reference to *hp140* and thus the semantics would be the same as in Fig. 14.

Our example model has only Choices as VSpecs, but the Autronica system has multiple examples of elements that are sets rather than singular choices. Such decision sets that represent repeated decisions on the same position in the VSpec tree are described by VClassifiers. Similar to ChoiceOccurrences that are typed Choices, we have VClassOccurrences that are typed VClassifiers. We appreciate that VClassifiers are not VTypes even though they represent reuse in some sense, but sets are not types. A type may have no occurrences, or several occurrences in different places in the VSpec tree.

II.5.5 Resolution Literals – describing subproducts

Once we have the VType with corresponding occurrences in the variability model, we may expect that there may be consequences of these changes in the associated resolution models and realization models.

What would be the VType counterpart in the resolution model?

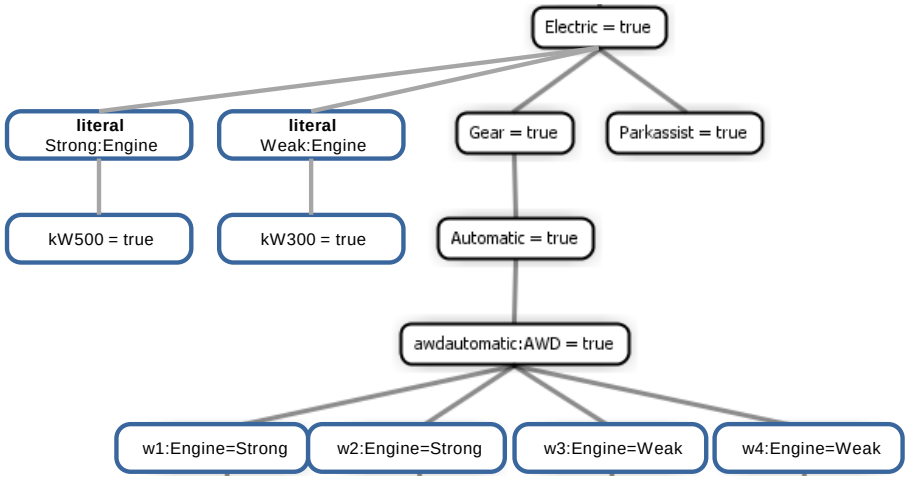

Fig. 16. Resolution literals

In Fig. 16 we show a resolution model of an imaginary electric car which has one engine for each wheel. We have defined two literals of the *Engine* type, one named *Strong* and one named *Weak*. The literals represent sub-products that have been fully resolved and named. In reality it is often the case that there are named sub-products that already exist and have product names. Thus such literals make the resolution models easier to read for the domain experts.

II.5.6 Staged Variation Points – realizing occurrences

Having seen that the VType has consequences for the resolution model, the next question is what consequences can be found in the realization model which describes the mapping between the variability model and the base model?

We have already reuse related to the realization layer since with fragment substitutions we can reuse replacement fragment types. Replacements represent general base model fragments that are cloned and inserted other places in the base model base.

Replacement fragment types do not correspond to VType directly and we find that with fragment substitutions as our main realization primitive we would need a hierarchical structure in the realization model to correspond to the hierarchy implied by occurrences of VTypes in the variability model. The "staged variation points" correspond closely with subtrees of the resolution model. They are not type symbols, but rather correspond to the expansion of occurrences (of VTypes and resolution literals).

In BVR (and CVL) variation points refer to a VSpec each. Materialization of a product is driven by the resolutions. They refer to VSpecs and trigger those variation points that refer to that same VSpec. A staged variation point refers to an occurrence of a VType.

The semantics of a staged variation point is to limit the universe of variation points from which to choose. The VSpec being materialized is an occurrence which refers to a VType. That VType has a definition containing a tree of VSpecs. The resolution element triggering the staged variation point has

a subtree of resolution elements that can *only* trigger variation points contained in the staged variation point.

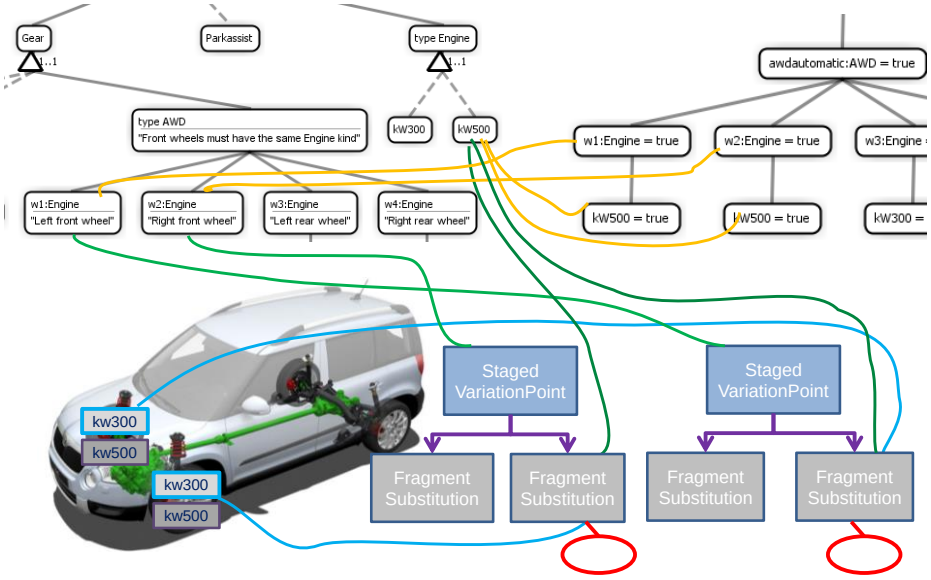

Fig. 17. Staged Variation Points example

In Fig. 17 we illustrate how staged variation points work. In the upper right we have the resolution model and we will concentrate on resolutions of *w1* and *w2*. *w1* is resolved to true and the rightmost staged variation point refers the *w1:Engine* choice on the very left in the VSpec model indicated by the green line. Now since the w1:Engine has been chosen we need to look into the *Engine* VType for what comes next, and the choice of the power of the engine comes next. For *w1* the resolution model indicates that *kw500* is chosen and this is also indicated by a yellow line from the resolution element to that of the VSpec model. The actual transformation of the base model is given by the variation points in the realization model, and we are now limited to the variation points enclosed by the staged variation point already found (the rightmost one). The rightmost fragment substitution of said staged variation point refers to the chosen *kw500* VSpec inside the *Engine* VType and thus this is the one that will be executed. The figure indicates that what it does is to remove the *kw300* option and leaving only the *kw500* engine option on the right wheel of the car.

In the very same way we may follow the resolution of *w2* and we find that due to the staged variation point for *w2* is the leftmost one, a different fragment substitution referring the *kw500* of the *Engine* VType will be executed for *w2* which is exactly what we need.

II.6 Discussion and relations to existing work

Here we discuss the new mechanisms and why they have not appeared just like this before.

II.6.1 The Target

Introducing targets was motivated by how the VSpec tree structure can be used to visualize and define restrictions to decisions. The more the tree structure is used to define the restrictions the more likely it is that there is a need to refer to the same target from different places in the tree.

Our example car in Fig. 13 can be described in another style as shown in Fig. 18 where the restrictions are given explicitly in constraints and the tree is very shallow. The two different styles, tree-oriented and constraint-oriented, can be used interchangeably and it may be personal style as well as the actual

variability that affects what style to choose. It is not in general the case that one style is easier or more comprehensible but constraints seem to need more familiarity with feature modeling [10].

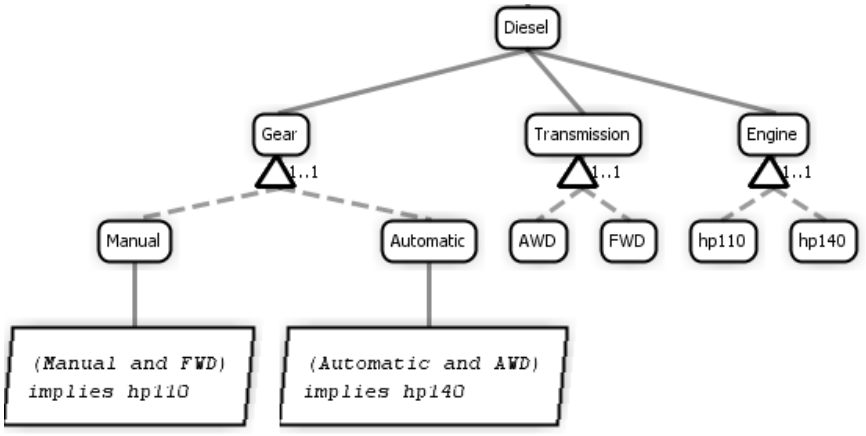

Fig. 18. The example car with explicit constraints

Given that a tree-oriented style is applied and there are duplications of target, why is this a novelty? It is a novelty because CVL does not have this concept and it is unclear whether other similar feature modeling notations support the distinction that we have named targets as distinguished from VSpecs (or features). Batory [11] and Czarnecki [12, 13] seem to solve duplication by renaming to uniqueness. The Atego OVM tool[5] implicitly forces the user into the style of using explicit constraints and thus circumvents the problem. OVM (Orthogonal Variability Model) [4] does not contain the general feature models of FODA [5].

VSpecs are distinct decision points and every VSpec is in fact unique due to the tree path to the root. Targets are also unique, but for a different reason. Targets are unique since they represent some substance that is singular. This substance needs not be base model specific, but it is often closely related to the base model. What makes this distinction essential is that explicit constraints talk about targets and not VSpecs. In Fig. 14 we have a variability model which is properly satisfied by (*Parkassist, Manual, AWD, hp140*) and by (*Parkassist, Automatic, AWD, hp140*) showing that *hp140* may refer to any of two distinct VSpecs.

II.6.2 The Type and its consequences

Introducing a type concept to BVR should come as no surprise. As pointed out in [14] concepts for reuse and structuring normally come very early in the evolution of a language. Since the feature models have a fairly long history [5] it may be somewhat surprising that type concepts for subtrees have not been introduced before. A type concept was introduced in the MoSiS CVL [1], and this was fairly similar to the one we introduce to BVR. The CVL Revised Submission [3] has a set of concepts related to "configurable units" that are related to our suggested VType concept, but those concepts were intended mainly for sub-product lines of larger size. The concept was also much related to how variabilities are visible from the outside and the inside of a product line definition.

Other notations have not introduced type concepts and this may indicate that the suggested notations were not really seen as modeling languages, but more as illustrations. Another explanation may be that type concepts do introduce some complexity that imply having to deal with some challenges.

[5] http://www.atego.com

One challenge is related to notation. The type must be defined and then used in a different place. In the singular world definition and usage were the same. VTypes must appear somewhere. We have chosen to place them within the VSpec tree, but it would also be attractive to be able to define VTypes in separate diagrams. A VType in fact defines a product line in its own right. Our Engine VType implied in Fig. 16 could contain much more than only horse power choice.

In the modeling language Clafer which has served as one of the inspiration sources of BVR, the type declarations must be on the topmost level [15] which in our example would have made no difference. Locally owned types, however, have been common in languages in the Simula/Algol tradition [16] for many years. The local ownership gives tighter encapsulation while the top ownership is semantically easier.

The usage occurrences refer to the type. How should this be depicted? We have chosen to use textual notation for this indicating the type following a colon. The colon is significant for showing that the element is indeed an occurrence of a VType.

Another challenge is related to how the VType and its occurrences are placed in the model at large. This has to do with what is often called scope or name space. We have defined that VTypes or VPackages (collections of VTypes) can be name spaces and thus occurrences of a VType *X* can only appear within the VType enclosing the definition of *X*, but VTypes may be nested. Similar to the discussion on targets, again names are significant because they designate something unique within a well-defined context.

Are targets and types related? Could we say that targets appearing in multiple VSpecs are in fact occurrences of a VType (named by the target name)? At first glance this may look promising, but they are conceptually different. The target is something invariant that the decisions mentioning it are talking about. A VType is a pattern for reuse, a tree structure of decisions representing a subproduct line. There are cases where the two concepts will coincide, but they should be kept distinct. While VTypes are defined explicitly and separately, we have chosen to let targets be defined implicitly through the names of VSpecs.

CVL already recognized types as it had VariableType which was quite elaborate and which also covered ReplacementFragmentType and ObjectType. Could VType be a specialized VariableType and the occurrences specialized variables? This may also be tempting, but variables are given values from the base model by the resolutions, while occurrences refer to patterns (VTypes) of the variability model.

II.6.3 The Note

The Note is about a significant element which has no direct significance in the language. Adding a note concept is an acknowledgement of the fact that there may very well be information that the user wants to associate closely with elements of the BVR model, but which is of no consequence to the BVR language or general BVR tooling.

Such additional information may be used for tracing, for expressing extra-functional properties or it may be pure comments. The text may be processed by proprietary tooling or by humans. Having no such mechanism made it necessary to accompany a CVL diagram with a textual description if it should be used by more than one person or more than one community.

Since variability modeling is oblivious to what varies, the Note can be more important than it might seem. The Note is where you can associate safety critical information with the variants and the possibilities. The Note is where you can contain traces to other models. The Note is where you can put requirements that are not connected to the variability model itself.

The Note will be significant for the tools doing analysis.

We foresee that once we have experimented with using notes in BVR, there will be recurring patterns of usage which may deserve special BVR constructs in the future, but at this point in time we find such constructs speculative.

II.6.4 The Reference

References in the BVR model are something that can be found in commercial tools like pure::variants[6]. A reference in the variability model is defined as a variable and as such it enhances the notion that variables hold base model values only. A *Vref* variable is resolved by a *VRefValueSpecification* where the pointers of the resolution model and the pointers of the variability model correspond in a commutative pattern.

Why are references necessary? They represent structure beyond the tree and this can represent dependencies which are hard to express transparently in explicit constraints.

In our motivation from the Autronica case our need for references came from describing an alternative product structure that overlaid the hierarchical physical structure of the configured system. We may say that our Autronica variability model is a very product-oriented (or base-oriented) variability model meaning that structures of the product was on purpose reflected in the variability model. The opposite would have been a property-oriented variability model where VSpecs would have represented more abstract choices such as "Focus on cost" vs. "Focus on response time".

II.7 Conclusions and Further Development

We have been motivated by needs of the use cases and found that the needs could be satisfied by introducing some fairly general new mechanisms. At the same time we have made the BVR language more compact than the original CVL language such that it serves a more focused purpose.

Our next step is to modify our CVL Tool Bundle to become a true BVR Tool Bundle to verify that the demo cases can more easily be expressed and maintained through the new language.

[6] http://www.pure-systems.de

II.8 REFERENCES

1. Haugen, O., B. Møller-Pedersen, J. Oldevik, G.K. Olsen, and A. Svendsen, *Adding Standardized Variability to Domain Specific Languages*, in *SPLC 2008*, B. Geppert and K. Pohl, Editors. 2008, IEEE Computer Society: Limerick, Ireland. p. 139-148.
2. Haugen, O., A. Wasowski, and K. Czarnecki, *CVL: common variability language*, in *Proceedings of the 17th International Software Product Line Conference*. 2013, ACM: Tokyo, Japan. p. 277-277.
3. OMG, *Common Variability Language (CVL)*. 2012, OMG: Needham, Massachusetts.
4. Pohl, K., G. Böckle, and F.J.v.d. Linden, *Software Product Line Engineering. Foundations, Principles and Techniques*. 2005: Springer. 468.
5. Kang, K., S. Cohen, J. Hess, W. Novak, and A. Peterson, *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. 1990, Software Engineering Institute, Carnegie Mellon University: Pittsburgh, PA.
6. Johansen, M.F., Ø. Haugen, and F. Fleurey, *An algorithm for generating t-wise covering arrays from large feature models*, in *SPLC '12 Proceedings of the 16th International Software Product Line Conference - Volume 1*. 2012, Association for Computing Machinery (ACM). p. 46-55.
7. Johansen, M.F., Ø. Haugen, F. Fleurey, A.G. Eldegaard, and T. Syversen, *Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines*, in *Model Driven Engineering Languages and Systems, 15th International Conference, MODELS 2012, Innsbruck, Austria, September 30–October 5, 2012. Proceedings*. 2012, Springer. p. 269-284.
8. Johansen, M.F., *Testing Product Lines of Industrial Size: Advancements in Combinatorial Interaction Testing*, in *Department of Informatics*. 2013, University of Oslo: Oslo. p. 231.
9. Berger, T., S. Stanciulescu, O. Ogaard, O. Haugen, B. Larsen, and A. Wasowski, *To Connect or Not to Connect: Experiences from Modeling Topological Variability*, in *SPLC 2014*. 2014: Florence. p. to be published.
10. Reinhartz-Berger, I., K. Figl, and Ø. Haugen, *Comprehensibility of Feature Models: Experimenting with CVL*, in *MODELS 2014*. 2014: Valencia. p. To be published.
11. Batory, D., *Feature Models, Grammars, and Propositional Formulas*, in *SPLC 2005*, H. Obbink and K. Pohl, Editors. 2005, Springer: Rennes. p. 7-20.
12. Czarnecki, K., S. Helsen, and U. Eisenecker, *Staged Configuration Using Feature Models.* Software Process Improvement and Practice, 2005. **10(2)**(Special issue on Software Variability: Process and Management): p. 143-169.
13. Czarnecki, K., S. Helsen, and U. Eisenecker, *Formalizing cardinality-based feature models and their specifications.* Software Process Improvement and Practice, 2005. **10(1)**: p. 7-29.
14. Haugen, O., *Domain-specific Languages and Standardization: Friends or Foes?*, in *Domain Engineering*, I. Reinhartz-Berger, et al., Editors. 2013, Springer: Heidelberg. p. 159-186.
15. Bąk, K., K. Czarnecki, and A. Wąsowski, *Feature and Meta-Models in Clafer: Mixed, Specialized, and Coupled*, in *Software Language Engineering*, B. Malloy, S. Staab, and M. van den Brand, Editors. 2011, Springer Berlin Heidelberg. p. 102-122.
16. Birtwistle, G.M., O.-J. Dahl, B. Myhrhaug, and K. Nygaard, *SIMULA BEGIN*. 1975, New York: Petrocelli/Charter.