The abstract syntax of structural VCL

Nuno Amálio and Pierre Kelsen
Laboratory for Advanced Software Systems
University of Luxembourg
6, rue R. Coudenhove-Kalergi
L-1359 Luxembourg

TR-LASSY-09-02

Contents

Chapter 1

Introduction

The visual contract language (VCL) has been designed to enable abstract specification of software systems visually and formally. We aim at obtaining a language that is intuitive, capable of expressing a large set of properties, enables precise specification and whose models can be formally analysed.

VCL has been designed so that the visual notations provided by VCL are used together with an underlying textual language that sits in the background. VCL embodies a flexible approach to semantics proposed in [1, 2], which we call *plug and play*. The formal textual specification language that sits in the background, which we call *target language*, must be accompanied by a VCL semantic model expressed in that language. We can plug different semantics expressed in different target languages (e.g. Z, Alloy, OCL). The result of a VCL specification is its semantics expressed in the target language. The aim is to express as much as possible visually and leave the underlying textual specification hidden as much as possible. This, however, is not always possible. Sometimes, the best solution is to write it down directly in the target language. Because of this, we consider that there is someone playing the rôle of the target language expert who is responsible for writing those properties that are not expressed visually and for doing dedicated tasks that require expertise with the target language tools.

This technical report presents the abstract syntax of the structural aspects of VCL, which comprise the notations of*structural* and *constraint* diagrams. The abstract syntax is formally defined using OO class *metamodels* in the formal language Alloy [3]. The alloy model defines the syntactic constructs of the language and describes well-formedness constraints. The metamodels described in Alloy were the basis for the concrete syntax metamodels implemented in VCL's tool: the *visual contract builder*[1].

To ease presentation, we overview the structure of our metamodels using UML class diagrams. The semantics of the class diagrams used here is as follows:

- Class diagrams have an Alloy semantics. Classes are defined as Alloy signatures; each denoting a set of atoms. Associations are represented as Alloy relations. Inheritance relations are defined using signature extensions.

- Compositions (associations with black diamonds) are normal relations, but they include an Alloy constraint forbidding sharing of the target objects.

The remainder of this technical report is as follows. Chapter 2 presents the abstract syntax of structural diagrams, and chapter 3 that of constraint diagrams.

[1] `http://vcl.gforge.uni.lu`

Chapter 2

Structural Diagrams

2.1 Overview

Structural diagrams (SDs) define the structures that make the state space of the system being specified. They enable the definition of the main problem domain concepts as blobs, their internal state as property edges and relations between concepts as relational edges.

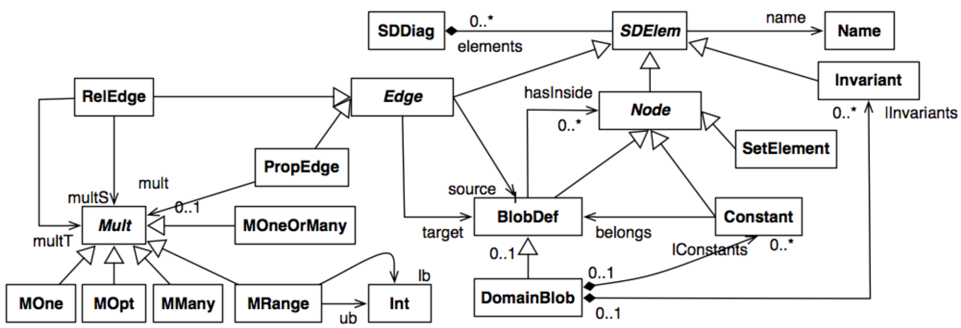

Figure 2.1: Metamodel describing the abstract syntax of structural diagrams.

Figure 2.1 presents metamodel of VCL SDs. It is as follows:

- Structural diagrams (*SDDiag*) are made up of a set of *elements* (*SDElem*). *SDElem*s have a name and are divided into *Node*, *Edge* and *Invariant*.
- *Node*, an abstract class, is specialised by *BlobDef*, *Constant* and *SetElement*. Blobs can have other nodes inside (*hasInside*). *DomainBlob* specialise *BlobDef* and have a set of local invariants (*lInvariants*) and local constants (*lConstants*). A *Constant belongs* to some blob; it is local if it's associated with some blob through relation *lConstanst* and global otherwise. *SetElements* can only exist inside *BlobDef*s; this is specified in Alloy model.
- *Invariants* can be global or local. They are local if associated with some blob through relation *lInvariants* and global otherwise.
- *Edge* represents edges whose nodes are *BlobDefs* (relations *source* and *target*). *Edge* is abstract and specialised by relational and property edges. *PropEdge*s have one multiplicity constraint (*mult*) attached to the target end. *RelEdge*s have two multiplicity constraints attached to both ends.
- Abstract class *Mult* represents all possible multiplicities; it is specialised by *MOne* (corresponds to 1), *MOpt* (optional or $0 \mathinner{..} 1$), *MMany* (many or $0 \mathinner{..} *$), *MOneOrMany* (at least one, or $1..*$) and *MRange* (a range, *lb* .. *ub*).

2.2 Alloy Models

This section presents the metamodels of VCL SDs in Alloy. The following describes two Alloy modules: the multiplicities module and the actual meta-model of structural diagrams. The latter imports the former. Multiplicities have been factored into a separate module because it is re-used in the metamodel of constraint diagrams.

2.2.1 Multplicities Module

The Alloy model for the multiplicities is as follows:

```
--====================================================================
-- Name: 'VCL_Mult'
--
-- Description:
--    + This is the module that defines multiplicities.
--====================================================================

module VCL_Mult

--======================================================================
-- Name: 'Mult' (Multiplicity)
--
-- Description:
--    + Defines what a multiplicity is.
--    + Multiplicities are attached to ends of edges.
-- Details:
--    + There are the folowing kinds of multiplicity: one, optional (0..1),
--    many (0..*), one or many (1..*), range (n1..n2).
--    + Multiplicities of kind range have a lower and an upper bound.
--======================================================================

abstract sig Mult {}

sig MOne, MOpt, MMany, MOneOrMany extends Mult {}

sig MRange extends Mult {
   -- lower and upper bound for 'range' multiplicities.
   lb, ub : lone Int
}{
   -- lower and upper bounds must be greater or equal than 0
   -- and 'ub' greater or equal than 'lb'.
   lb >= 0 && ub >= lb
}
```

2.2.2 Structural Diagrams Module

The next Alloy model describes all the main concepts of VCL. It imports the Alloy model for multiplicities. Together these two Alloy models describe the meta-model of VCL structural diagrams in Alloy.

```
--=======================================================================
-- Name: 'VCL_SD'
--
-- Description:
```

```
--    + This module meta-model of VCL structural diagrams.
--========================================================================

module VCL_SD

open VCL_Mult as m

--=========================================================================
-- Name: 'Name'
--
-- Description:
--    + Introduces set of labels to be attached to nodes and edges
--=========================================================================

-- Signature of all names
sig Name {}

--=========================================================================
-- Name: 'SDElem'
--
-- Description:
--    + Introduces the labelled structural diagram element.
--    + To be extended by 'BlobDef', 'Object' and 'Edge'.
--
--   -----------    0..1------
--    |SDElem   |------->|Name|
--   -----------   name ------
--=========================================================================

--
-- A modelling element may be labelled with a Name.
-- Modelling elements are subdivided into 'Node' and 'Edge'
abstract sig SDElem {
   name  : Name
}

--============================================================================
-- Name: 'Node'
--
-- Description:
--    + Nodes of VCL graphs structures.
--    + To be extended by blob and object.
--============================================================================

abstract sig Node extends SDElem {
}

--=========================================================================
-- Name: 'BlobDef' (Blob Definitions)
--
-- Description:
--    + Defines a global blob definition.
--    + It's characterised by inside property.
```

```
--
--   ---------         0..*-----------
--   |BlobDef|-------------->| Node   |
--   ---------    hasInside ----------
--
--=========================================================================

sig BlobDef extends Node {
   hasInside      : set Node
}
{
   -- A blob def may have inside either blob defs or set elements
   hasInside in (BlobDef+SetElement)
}

--
-- The following defines what it means for VCL structures to be well-formed
-- regarding the 'inside' property.
--
-- The graph representing the 'inside' relation should be acyclic.
fact acyclicInside {
   no ^hasInside & iden
}

--
-- The transitive constructions on the blob relation are unnecessary because
-- they can be obtained through the transitive closure
fact insideTransitiveIsRedundant {
   all n1, n2, n3 : Node | n1->n2 in hasInside && n3 in n2.^hasInside
      => !(n1->n3 in hasInside)
}

--
-- This function gets all property edges of some blobDef
fun getPropEdgesOfBlob [blob : BlobDef] :  PropEdge {
   {pe1 : PropEdge | pe1.source = blob}
}


--=========================================================================
-- Name: 'DomainBlob' (Domain Blob)
--
-- Description:
--    + Defines a global blob definition.
--    + And by a set of local constants and local invariants.
--
--   -------------
--   |Domain Blob|
--   -------------
--     | |              0..*-----------
--     | |------------------>|Invariant|
--     |         lInvariants -----------
--     |                0..*-----------
--     |-------------------->|Constant |
```

```
--              lConstants  -----------
--
--============================================================================

sig DomainBlob extends BlobDef {
   lInvariants : set Invariant,
   lConstants  : set Constant
}
{
   -- A local constant cannot belong to the blob for which it is defined
   -- (No Circular definition)
   this != lConstants.belongs
}

--
-- Each 'DomainBlob' has its own set of local invariants
-- Or local constants are not shared.
fact LInvariantsNotShared {
   all i : Invariant | (some lInvariants.i)
      =>  one lInvariants.i
}

--
-- Each 'DomainBlob' has its own set of local constants
-- Or local constants are not shared.
fact LConstantsNotShared {
   all c : Constant | (some lConstants.c)
      =>  one lConstants.c
}

--
-- Each domain blob can contain other domain blobs obly
-- and they can be inside of domin blobs only.
fact DBlobHasDBlobsInside {
   all db : DomainBlob |
      db.hasInside in DomainBlob && hasInside.db in DomainBlob
}

--==============================================================================
-- Name: 'SetElement'
--
-- Description:
--    + The elements that can be inside a blob (defined as enumeration).
--    + A set Element is a 'Object'.
--=========================================================================

sig SetElement extends Node {
}

--
-- A set element must be inside one blob (one blob only)
-- This Blob must not be a domain blob
fact SetElementInsideOneBlob {
   all se : SetElement | one bd : BlobDef | se in bd.hasInside
```

```
}

--
-- Set elements have unique names
fact SetElementNamesAreUnique {
   all n : Name | some (n.~name & SetElement) => one  n.~name
}

--===========================================================================
-- Name: ’Edge’
--
-- Description:
--    + Defines a binary edges as connecting two nodes.
--    + This is to be extended by ’OntoEdges’ and ’BlobEdge’.
--===========================================================================
abstract sig Edge extends SDElem {
   source : BlobDef,
   target : BlobDef
}

--=========================================================================
-- Name: ’PropEdge’ (Property Edges)

-- Description:
--    + Property edges define properties of blobs.
--    + They relate one blob (having property) to another (type of property).
--    + A property edge has a ’BlobDef’ as target.
--    + A property edge may have a multiplicity.
--
--   ----------          0..*---------
--   |PropEdge|------------->|BlobDef|
--   ----------       target ---------
--=========================================================================

sig PropEdge extends Edge {
   mult : lone Mult
}
{
   -- a ’PropEdge’ should not be ’onto’ itself
   source != target

   -- a property edge should not be onto any of the blobs inside
   not (target in (source.^hasInside))
}
   -- A Property edge has a multiplicity constraint. If none is
   -- explicitly provided then multiplicity ’1’ is assumed.

--
-- Each ’BlobDef’ has its own set of property edges
-- Or property edges are not shared.
fact propEdgesNotShared {
   all pe : PropEdge | (some pe.source)
      =>  one pe.source
}
```

```

--
-- Names of property edges in the scope of a 'BlobDef'must be unique
--
fact PropEdgeNamesAreUnique {
   all pe1, pe2 : PropEdge | all b : BlobDef |
      pe1.name = pe2.name && (pe1+pe2) in getPropEdgesOfBlob [b]
      => pe1 = pe2
}

--=============================================================
-- Name: 'RelEdge' (Relational Edge)
--
-- Description:
--    + Blob relational edges extend blob edges.
--    + They are binary edges connecting blobs.
--    + Relational Edges have multiplicities
--=============================================================

sig RelEdge extends Edge {
    multS, multT : Mult,
}

--
-- Relational edges names must be unique
--
fact RelEdgeNamesAreUnique {
   all n : Name | some (n.~name & RelEdge) => one  n.~name
}

--=========================================================================
-- Name: 'Constant'
--
-- Description:
--    + 'Constant' are objects.
--    + Constants are members of a blob; relation 'belongsTo'.
--    + Constants can be connected to some local blob (local constant).
--    + Or they can stand alone (global constant)
--=========================================================================

sig Constant extends Node {
   belongs : BlobDef
}

--
-- Constants have unique names
fact ConstantNamesAreUnique {
   all n : Name | some (n.~name & Constant) => one  n.~name
}


--=============================================================
-- Name: 'Invariant'
--
```

```
-- Description:
--    + Invariants can be connected to some blob (local invariant)
--    + Or they can stand alone (global invariants)
--=============================================================

sig Invariant extends SDElem {
}

--
-- Blob defs and invariant names must be unique
fact BlobAndInvariantNamesAreUnique {
    all n : (BlobDef+Invariant).name | one n.~name
}
```

Chapter 3

Constraint Diagrams

3.1 Overview

The metamodel of CntDs is divided in: predicate elements, constraint reference expressions, declarations and communication edges, and constraint diagrams.

3.1.1 Predicate Elements.

Figure 3.1 presents this part of the metamodel that describes the syntactic constructions involving objects, blobs and edges that form the CntD's predicate. It is as follows:

- Predicate elements (*CntPElem*) are divided into *CntNode* and *CntEdge*. *CntPElem* includes a designator, which is either a name or some textual expression (e.g. $balance - amount?$).
- *CntNode* is divided into *CntBlob* and *CntObject*. *CntBlob* has inside property (*hasInside*); it is specialised by *ShadedBlob* to represent blobs that are shaded.
- *CntEdge* is divided into *RelCEdge* and *ValEdge*. *RelCEdges* (relational constrained edge) can have multiplicities associated with target and source nodes. *ValEdge* (value edge) has an operator (equality by default); its subclasses represent operators that can be used (not equal, greater than, etc.).

3.1.2 Constraint Reference Expressions.

Figure 3.2 presents the metamodel for predicates made of constraint reference expressions. Metamodel is as follows:

- Constraint reference expressions (*CntRExp*) are divided into quantified (*CntRQExp*) and propositional (*CntRPropExp*) constraint expressions.
- *CntRQExp* quantifies a propositional formula (*exp* relation); it is is divided into universal (*CntRAll*) and existential (*CntRExists*) expressions.
- *CntRPropExp* is divided into terms (*CntRTerm*) and combinations of terms (*CntRComb*). *CntRComb* defines combinations using conjunction, disjunction and implication; it has terms on left- and right- hand sides (*lhs*, *rhs*).
- *CntRTerm* is divided into atoms (*CntRAtom*) and negations (*CntReg*). Negations comprise *atom* being negated. Atoms have a name.

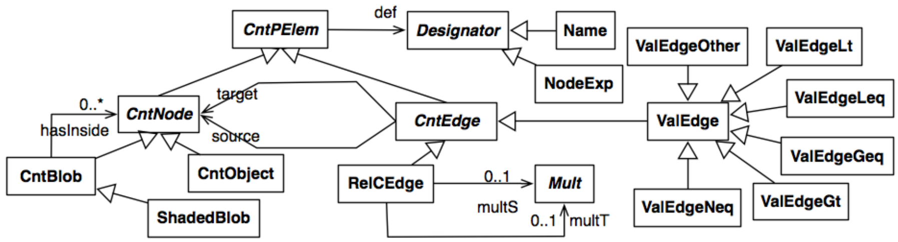

Figure 3.1: Metamodel describing the syntax of constraint diagrams, predicate elements.

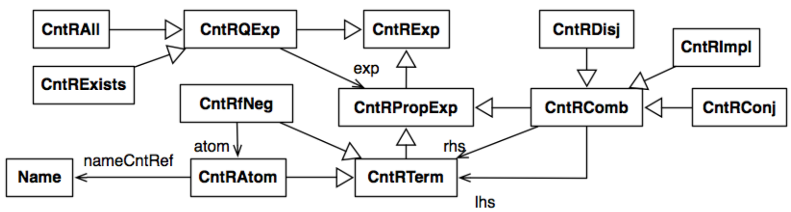

Figure 3.2: Metamodel describing the syntax of constraint reference expressions.

3.1.3 Declarations and communication edges.

The declarations compartment introduces names that take part in a constraint's description, along with constraints being imported. Communication edges facilitate communication between current and imported constraint diagrams. Figure 3.3 presents the metamodel; it is as follows:

- Declarations compartment elements (*CntDeclElem*) have a *name*. They are divided into *Decl* and *CntRef*, which represents constraints being imported.
- *Decl* is divided into blob (*BlobDecl*) and object (*ObjDecl*) declarations.
- *CommNode* and *CommEdge* deal with communication edges. A communication node can either be a declaration element, a node in the predicate, or a constraint reference expression. *CommEdge* has a name representing the channel to send or receive information, and comprises two nodes indicating origin and destination of communication (*from* and *to*). There are two kinds of edges: those involved in CntDs with predicate elements (*EdgeElems*), and those involved in a constraint reference expression (*EdgeCntRExp*).

Several important well-formedness constraints are expressed in Alloy. *EdgeElems* must represent those edges that connect predicate nodes (blobs or objects) to imported constraints. *EdgeCntRExp*, on the other hand, must represent those edges that connect objects and blobs from declarations into a constraint reference expression in predicate.

3.1.4 Constraint Diagrams as a whole.

The different syntactic structures presented above are put together in Fig. 3.4; the metamodel is as follows:

- A constraint diagram (*CntDiag*) comprises a name (*cntName*), *declarations*, *predicate* and a set of communication edges (*commEdges*).
- *CntPredicate* can either be an elements predicate (*ElemsP*) or a constraint reference predicate (*CntRefP*). *ElemsP* is made up of a set of predicate elements; *CntRefP* is made up of a constraint reference expression.

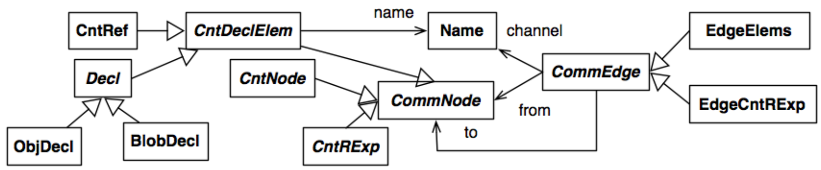

Figure 3.3: Metamodel describing the syntax of declarations and communication edges.

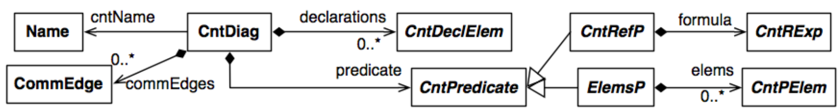

Figure 3.4: Metamodel describing the syntax of constraint diagrams as a whole.

3.2 Alloy Models

In this section we present the abstract syntax of constraint diagrams in Alloy. The description is composed of the following Alloy modules: predicate elements, constraint reference expressions, declarations and constraint diagrams.

3.2.1 Predicate Elements

The Alloy model for the multiplicities is as follows:

```
--=======================================================================
-- Name: 'VCL_CntD_PElems'
--
-- Description:
--     + This module defines structures used in predicate compartment of
--         VCL blob constraint diagrams.
--========================================================================

module VCL_CntD_PElems

open VCL_Mult as m
open VCL_CntD_Common as c

--=======================================================================
-- Name: 'DefExp' (Node Defining Expression)
--
-- Description:
--    + Represents a textual expression defining the node (eg. amount? + balance)
--========================================================================

sig DefExp {}

--=======================================================================
-- Name: 'Designator' (Designator)
--
-- Description:
--    + Defines a designator in constraint diagram predicate.
--    + A designator can be a name or some designating expression.
```

```
--==========================================================================

sig Designator in Name+DefExp {}

--================================================================================
-- Name: 'CntPElem ' (Designator)
--
-- Description:
--    + An element that can appear in predicate comaprtment of constraint diagram.
--    + An element is identified by a designator.
--
--    --------------       1 -------------
--    |CntPElem |-------->|Designator |
--    --------------     def -------------
--================================================================================

abstract sig CntPElem  {
   def     : Designator
}

--==============================================================================
-- Name: 'CntNode'
--
-- Description:
--     + Nodes of VCL constraint diagram.
--     + To be extended by blobs and objects used in Cnt Diagrams.
--==============================================================================

abstract sig CntNode  extends CntPElem {}

--
-- 'NodeDefExp' are not shared across nodes.
--fact NodeDefExpNotShared {
--  all nde : NodeDefExp | one def.nde
--}

--=============================================================================
-- Name: 'CntEdge'
--
-- Description:
--     + Defines a binary edges as connecting two Cntnodes.
--     + This is to be extended by 'ValEdge' and 'RelCEdge'.
--=============================================================================

abstract sig CntEdge extends CntPElem {
   source : CntNode,
   target : CntNode
}{
   -- The designator of an edge must be a name
   def in Name
}

--==========================================================================
-- Name: 'CntBlob' (Constraint Blob)
```

```
--
-- Description:
--    + Blobs that may occur in blob constraint diagrams.
--    + These are formed by refering to existing blobs (in a SD),
--       enclosing other blobs within, or by adding value edges.
--    + Blobs have the special inside property enabling other nodes inside.
--    + Inside indicates nodes that a blob encloses.
--
--   ------------------
--   |CntBlob         |      0..*---------
--   |   ------------ |--------->|CntNode|
--   |   |ShadedBlob| |          ---------
--   |   ------------ |
--   ------------------
--
--==========================================================================

sig CntBlob extends CntNode {
   inside  : set CntNode
}

--
-- The following defines what it means for VCL structures to be well-formed
-- regarding the 'inside' property.
--
-- The graph representing the 'inside' relation should be acyclic.
fact acyclicInside {
   no ^inside & iden
}

--
-- The transitive constructions on the blob relation are unnecessary because
-- they can be obtained through the transitive closure
fact insideTransitiveIsRedundant {
   all n1, n2 : CntBlob, n3 : CntNode | n1->n2 in inside && n3 in n2.^inside
      => !(n1->n3 in inside)
}

--==========================================================================
-- Name: 'ShadedBlob' (Shaded Blob)
--
-- Description:
--    + Represents those Blobs that are shaded.
--
--==========================================================================

sig ShadedBlob extends CntBlob {}

--==========================================================================
-- Name: 'CntObj' (Constraint Object)
--
-- Description:
--    + Objects that may occur in constraint diagrams.
--===========================================================================
```

```

sig CntObj extends CntNode {
}

--=======================================================================
-- Name: 'ValEdge' (edge values connected to objects or blobs)
--
-- Description:
--    + Connects blobs and objects to other blobs and objects.
--    + A value edge includes an operator by default its '='.
--========================================================================

sig ValEdge extends CntEdge {
}{
   -- a 'ValEdge' should not be 'onto' itself
   source != target
}

--
-- This function gets all property edges of some CntNode.
--
fun getValEdgesOfCntNode [n : CntNode] :  ValEdge {
   {ve1 : ValEdge | ve1.source = n}
}

--
-- Nodes that are the target of a value edge must be inside a blob
-- if the blob they have as target is also inside a blob
fact targetsOfValEdgeInsideIfSourceNodeAlsoInside {
   all n : CntNode | some inside.n =>
       (all ve : getValEdgesOfCntNode [n] | inside.(ve.target) = inside.n)
}

--
-- Nodes cannot have other nodes that they have inside as targets
fact NoValEdgeTargetInsideOfSource {
   all ve : ValEdge | !(ve.target in (ve.source).inside)
}

--=======================================================================
-- Name: 'ValEdge' (edge values connected to objects or blobs)
--
-- Description:
--    + The extensions of 'ValEdge' representing different operators
--========================================================================

sig ValEdgeNeq, ValEdgeGt, ValEdgeGeq, ValEdgeLeq,
ValEdgeLt, ValEdgeOther  extends ValEdge {}


--========================================================================
-- Name: 'RelCEdge' (Relational constrained edge)
--
-- Description:
```

```
--   + Defines relational constrained Edge ('RelCEdge').
--   + Used to define relations between blob references
--   + and links between objects.
--   + These relations may refine multiplicity constraints.
--========================================================================

sig RelCEdge extends CntEdge {
    multEndS, multEndT : lone Mult
}
```

The next Alloy model describes all the main concepts of VCL. It imports the Alloy model for multiplicities. Together these two Alloy models describe the meta-model of VCL structural diagrams in Alloy.

3.2.2 Constraint Reference Expressions

```
--=======================================================================
-- Name: 'VCL_CntD_PCntRefExp'
--
-- Description:
--    + Defines prededicate expressions made up of constraints references.
--========================================================================

--------------------------------
-- The grammar is as follows:
-- CntRAtom ::= Name
-- CntRTerm ::= CntRAtom | not CntRAtom
-- CntRPropExp   ::= CntRTerm |
--                    CntRTerm implies CntRTerm |
--                    CntRTerm and CntRTerm |
--                    CntRTerm or CntRTerm
-- CntRQExp ::= all CntRComb | exists CntRComb
-- CntRExp ::= CntRQExp | CntRPropExp
---------------------------------

module VCL_CntD_PCntRExp

open VCL_CntD_Common as c

--=======================================================================
-- Name: 'CntRExp' (Constraint Reference Expression)
--
-- Description:
--   + Abstract to be extended by actual expressions (propositions or quantified expressions)
--========================================================================

abstract sig CntRExp {}

--=======================================================================
-- Name: 'CntRPropExp' (Constraint Reference Propositional Expression)
--
-- Description:
--   + Abstract to be extended by actual expressions (atom, neg, comp)
--========================================================================
```

```

abstract sig CntRPropExp extends CntRExp {}


--=======================================================================
-- Name: 'CntRTerm' (Constraint Reference Term)
--
-- Description:
--   + Abstract to be extended by atom or negation
--========================================================================

abstract sig CntRTerm extends CntRPropExp {}

--=======================================================================
-- Name: ' CntRAtom' (Constraint Reference Atom)
--
-- Description:
--   + Constraint reference atom is made of a name (name of constraint)
--========================================================================

sig CntRAtom extends CntRTerm {
   nameCntR : Name
}

--=======================================================================
-- Name: 'CntRNeg' (Constraint Reference Negation)
--
-- Description:
--   + Constraint reference negation (negates an atom)
--========================================================================

sig CntRNeg extends CntRTerm {
   term : CntRAtom
}

--=======================================================================
-- Name: 'CntRComb' (Constraint Reference Combination)
--
-- Description:
--   + Combines two constraint references using some logical operator.
--========================================================================

abstract sig CntRComb extends CntRPropExp  {
  lhs : CntRTerm,
  rhs : CntRTerm
}

--=======================================================================
-- Name: 'CntRConj' (Constraint Reference Conjunction)
--
-- Description:
--   + Combines two constraint references using conjunction.
--========================================================================

```

```
sig  CntRConj extends CntRComb {}


--=======================================================================
-- Name: 'CntRDisj' (Constraint Reference Disjunction)
--
-- Description:
--   + Combines two constraint references using disjunction.
--========================================================================

sig  CntRDisj extends CntRComb {}

--=======================================================================
-- Name: 'CntRImpl' (Constraint Reference Implication)
--
-- Description:
--   + Combines two constraint references using implication.
--========================================================================

sig  CntRImpl extends CntRComb {}

--=======================================================================
-- Name: 'CntRQExp' (Constraint Reference Quantified expression)
--
-- Description:
--   + Defines an expression that includes a quantifier.
--========================================================================
abstract sig  CntRQExp extends CntRExp {
   exp : CntRPropExp
}

sig  CntRAll extends  CntRQExp {}

sig  CntRExists extends  CntRQExp {}

```

3.2.3 Declarations

```
--=======================================================================
-- Name: 'VCL_CntD_Decl'
--
-- Description:
--    + Defines declarations compartment of a constraint diagram.
--    + Enables importing of constraints and declaration of inpus and outpus.
--========================================================================


module VCL_CntD_Decl

open VCL_CntD_Common as c

--=======================================================================
-- Name: 'CntDeclElem' (Constraint Declarations Element)
--
```

```
-- Description:
--   + Element of declarations compartment (input, output, constraint ref).
--========================================================================

abstract sig CntDeclElem {
   name : Name
}

--=======================================================================
-- Name: 'Decl' (A declaration)
--
-- Description:
--    + Defines a declaration.
--    + Comprises a 'name' and a 'belongs'.
--========================================================================

abstract sig Decl extends CntDeclElem {
   -- A declaration indicates blob it belongs to (a Name)
   belongs : Name
}{
   -- 'name' and 'belongs' are different
   name != belongs
}

--=======================================================================
-- Name: 'BlobDecl' (A blob declaration)
--
-- Description:
--    + Defines a blob declaration.
--========================================================================

sig BlobDecl extends Decl {
}{
}


--=======================================================================
-- Name: 'ObjDecl' (An object declaration)
--
-- Description:
--    + Defines an object declaration.
--========================================================================

sig ObjDecl extends Decl {
}{
}

--=======================================================================
-- Name: 'CntRef' (Defines a constraint reference)
--
-- Description:
--    + Defines a constraint reference.
--========================================================================

```

```
sig CntRef extends CntDeclElem {
}
```

3.2.4 Constraint Diagrams

```
--========================================================================
-- Name: 'VCL_CntD'
--
-- Description:
--    + This module defines what a constraint diagram is.
--=========================================================================

module VCL_CntD

open VCL_CntD_Decl as d
open VCL_CntD_PElems as e
open VCL_CntD_PCntRefExp as ce

--=========================================================================
-- Name: 'CntPred' (Constraint Predicate)
--
-- Description:
--   + Defines a constraint predicate.
--   + Abstract to be specialised by 'ElementsP' or 'CntRefExpP'
--=========================================================================

abstract sig CntPred {}{
   -- 'CntPred' must be part of a Constraint Diagram
   this in CntDiag.predicate
}

--=========================================================================
-- Name: 'ElemsP' (Elements Predicate)
--
-- Description:
--   + Defines an elements predicate.
--=========================================================================

sig ElemsP extends CntPred {
   elems : set CntPElem
}

--
-- Predicate elements are not shared across constraint diagrams.
--
fact ElemsPNotShared {
  all e : CntPElem | e in ElemsP.elems => one elems.e
}

--=========================================================================
-- Name: 'CntRefP' (CntRef Predicate)
--
-- Description:
--   + Defines a predicate of type constraint reference expression.
```

```
--   + Just a CntRefExp
--=========================================================================

sig CntRefP extends CntPred {
   formula : CntRExp
}

--
-- CntRef Expressions are not across constrain diagrams
--
fact CntRefPNotShared {
  all e : CntRExp | e in CntRefP.formula => one formula.e
}

--=========================================================================
-- Name: ' CommNode'
--
-- Description:
--   + Defines a communication node: 'CntNode' or 'ConstraintRef'.
--=========================================================================

sig CommNode in CntNode+CntDeclElem+CntRExp {}

--=========================================================================
-- Name: 'CommEdge'
--
-- Description:
--   + Defines a communication edg.
--   + A communication edge has a channel.
--   + Abstract to be specialised by 'CommEdgeElems' and 'CommEdgeCntRefExp'
--=========================================================================

abstract sig CommEdge {
   channel : Name,
   from    : CommNode,
   to      : CommNode
}{
   -- A comm Edge must belong to a constraint diagram
   this in CntDiag.commEdges
}

--
-- Communication edges are not shared
--
fact CommEdgesNotShared {
   all ce : CommEdge | one commEdges.ce
}


--=========================================================================
-- Name: 'CommEdgeElems'
--
-- Description:
--   + Defines a communication edge liking CntRef to CntNode
```

```
--=========================================================================

sig CommEdgeElems extends CommEdge {
}{
   -- from to represent link between 'CntNode' and 'CntRef'
   --from in CntNode => to in CntRef
   one ((from+to) & CntRef)
   one ((from+to) & CntNode)
}

--
-- Communication edges must link elements from declarations and predicate of CntD
--
fact NodesOfCommEdgesInCntD {
   all ce : CommEdge |
         (ce.from+ce.to) in
            (commEdges.ce).(declarations+predicate.elems+predicate.formula)
}

--=========================================================================
-- Name: 'CommEdgeCntRefExp'
--
-- Description:
--   + Defines a communication edge a declaration to a CntRefExp
--=========================================================================

sig CommEdgeCntRefExp extends CommEdge {
}{
   -- From must be a delcaration and to a 'CntRefExp'
   from in Decl && to in CntRExp
}

--
-- Communication edges must link elements from declarations and predicate of CntD
--
fact NodesOfCommEdgesInCntD {
   all ce : CommEdgeElems |
      ce in CntDiag.commEdges =>
         (ce.from+ce.to) in (commEdges.ce).(declarations+predicate.elems)
}

--=========================================================================
-- Name: ' CntDiag'
--
-- Description:
--   + Defines what a constraint diagram is.
--   + A constraint diagram comprises:
--      + a name, identifying name of constraint being defined.
--      + A declarations compartment including inputs, outputs and constraint imports
--      + A predicate which may be of type elements or Cnt Ref Exp.
--=========================================================================

sig CntDiag {
   cntName      : Name,
```

```
   declarations : set CntDeclElem,
   predicate    : CntPred,
   commEdges    : set CommEdge
}{
   --If predicate is of type 'elements' then 'commEdges' must also be of this type
   predicate in ElemsP => commEdges in CommEdgeElems
   --If predicate is of type 'CntRefExpP' then 'commEdges' must also be of this type
   predicate in CntRefP => commEdges in CommEdgeCntRefExp
}

--
-- Function to retrieve constraint imports
--
fun getImports [cd : CntDiag] :  CntRef {
   (cd.declarations) & CntRef
}

--
-- A constraint diagram may not import itself
--
fact SelfImportingNotAllowed {
   all cd : CntDiag | all cref : getImports[cd] | cd.cntName != cref.name
}

--
-- Each constraint diagram has its own declaration elements.
--
fact DeclElementsNotShared {
   all de : CntDeclElem | de in CntDiag.declarations => one declarations.de
}

--
-- Names of declarations in the scope of a constraint diagram must be unique
--
fact DeclNamesAreUnique {
   all cd : CntDiag |  all n : (cd.declarations).name | one (name.n & (cd.declarations))
}

--
-- Function to retrieve node declarations
--
fun getNodeDecls [cd : CntDiag] :  Decl {
   (cd.declarations) & Decl
}
```

References

[1] Amálio, N. *Generative frameworks for rigorous model-driven development.* Ph.D. thesis, Dept. Computer Science, Univ. of York (2007)

[2] Amálio, N., Polack, F., Stepney, S. Frameworks based on templates for rigorous model-driven development. *ENTCS*, 191:3–23 (2007)

[3] Jackson, D. *Software Abstractions: logic, lanaguage, and analysis.* MIT Press (2006)