The Type System of VCL

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

TR-LASSY-12-11
# Contents

## 1 Introduction

1.1 Background: The Visual Contract Language (VCL)  
1.1.1 VCL Diagrams  
Package Diagrams  
Structural Diagrams  
Behaviour Diagrams  
Assertion Diagrams  
Contract Diagrams  
1.1.2 Semantics  
1.2 Outline

## 2 Syntax

2.1 Metamodels  
2.1.1 Package Diagrams  
2.1.2 Common  
2.1.3 Structural Diagrams  
2.1.4 Common Assertion and Contract Diagrams  
2.1.5 Assertion Diagrams  
2.2 Grammars

## 3 Type System

3.1 Types and Environments  
3.2 Basic Rules  
3.3 Common Rules  
3.4 Package Diagrams  
3.5 Structural Diagrams  
3.6 Assertion Diagrams  
3.7 VCL Models

## References

## A Auxiliary Definitions

A.1 Environment Operators
A.2 Predicates  
A.2.1 Predicate Acyclic  
A.2.2 Predicates IsPEP and IsPEM
| A.2.3 Predicate NoClashes               | 46 |
| A.2.4 Predicate PkgOnce               | 47 |
| A.2.5 Predicate AcyclicPkg             | 47 |
| A.3  Auxiliary Functions              | 47 |
| A.3.1 Functions to extract information from PDs | 47 |
| A.3.2 Function buildDE               | 47 |
| A.3.3 Function getGType               | 47 |
| A.3.4 Function superTy                | 48 |
| A.3.5 Functions producing variable environments (VEs) | 48 |
| A.3.6 Function transferBindingsForPkgInc | 48 |
| A.3.7 Function getDK                  | 49 |
| A.3.8 Functions to extract information from ADs | 49 |
| A.3.9 Functions for AD lookup         | 50 |
| A.3.10 Functions for substitutions    | 51 |
| A.3.11 Function getSIdFrScalarOrCollection | 51 |
| B  Alloy Metamodels of VCL Diagrams   | 52 |
| B.1 Package Diagrams                 | 52 |
| B.2 Common                            | 55 |
| B.3 Structural Diagrams              | 64 |
| B.4 Common Assertion and Contract Diagrams | 71 |
| B.5 Assertion Diagrams               | 76 |
Chapter 1

Introduction

This document presents a type system for the Visual Contract Language (VCL) \cite{AK10,AKMG10}. This formalises a typed object-oriented system with subtyping. This type system has been implemented in the VCL tool, the Visual Contract Builder \cite{AGK11}. The following gives some background on VCL and an outline of the overall document.

1.1 Background: The Visual Contract Language (VCL)

VCL \cite{AK10,AKMG10,AGK11} is a formal language designed for the abstract description of software systems. Its modelling paradigms are set theory, object-orientation and design-by-contract (pre- and post-conditions). VCL’s distinguishing features are its capacity to describe predicates visually and its approach to behavioural modelling based on design by contract.

VCL’s semantics is based on set theory. Its semantics definition takes a translational approach. Currently, VCL has a Z semantics: VCL diagrams are mapped to ZOO \cite{APS05,Amá07}, a semantic domain of object orientation for the language Z \cite{Spi92,ISO02}.

1.1.1 VCL Diagrams

VCL’s diagram suite comprises package, structural, behaviour, assertion and contract diagrams.

Package Diagrams

Package diagrams (PDs) define VCL packages, coarse-grained modules representing concerns. Packages are represented as clouds because they define a world of their own. Sample PDs from the secure simple bank case study \cite{Amá11} are given in Fig. 1.1. PDs are as follows:

- The package being defined or the current package is represented in bold. The current package can either be defined as a container (symbol $\Box$) or ensemble (symbol $\ast$). Container packages merely contain sets and their local definitions. Ensemble packages have a global identity; they may include relations between sets and global invariants and operations. In Fig. 1.1, packages CommonTypes and RolesAndTasksBank are containers; all others are ensembles.

\url{http://vcl.gforge.uni.lu/}
A current package may define to use elements defined in other packages. *Uses* edges are represented with a hollow headed arrows. In Figs. 1.1b and 1.1c, packages Bank and Authentication use package CommonTypes.

The current package can incorporate other packages, which means that the current package includes the structures of incorporated packages plus some structures of its own. Package incorporation is expressed using *enclosure*. In Fig. 1.1h, package SecBank incorporates packages BankWithJI, SecForBank and TransactionsSwitch.

To resolve conflicts with package incorporations, it is possible to express in a PD conflict resolution dependencies using edges. There two kinds of such edges: *overrides* and *merges*. Override edges says that certain sets in the source package override those with the same name in the target package. Merge edges say that certain specified sets with the same name from the linked packages are to be merged. Figure 1.1f says that set User of package Authentication overrides User of AccessControl.

The current package may extend incorporated sets. This is specified using an extends list. In Fig. 1.1g, package SecForBank extends incorporated set User.

### Structural Diagrams

Structural diagrams (SDs) define the structures that make the state space of a VCL package. Figure 1.2 gives sample SDs from secure simple bank. SDs are as follows:

- SDs define two kinds of sets: *value* and *class*. Classes are distinguished from their value counterparts through a bold line. In Fig. 1.2a all sets are value. In SDs of Fig. 1.2, Customer, Account, User and Session are class sets; all others are value sets.

- Sets that include symbol $\circ$ are *definitional*. This means that they are defined by what they enclose. Sets that include symbol $\circ$ followed by $\leftrightarrow$ are *derived*. This means that they are
defined from primitive entities of the model. In Figs. 1.2b and 1.2c, sets CustType, AccType, LoginResult and UserStatus are enumerations defined by indicating their possible values. In Fig. 1.2a, TimeNat is a derived set defined from the set of natural numbers Nat.

- **Reference** sets include symbol ↑; they are used to refer to sets defined in used packages as defined in the PD (those sets are visible). SDs of Figs. 1.2b and 1.2c have reference sets referring to Name and TimeNat defined in package CommonTypes (alias CT).

- Edges with circled labels are relation edges. They define binary relations between sets. Directed edges are property-edges. They define state properties (attributes or fields) possessed by all objects of some set. In Figs. 1.2b and 1.2c, Holds and HasSession are relation-edges, and all outgoing edges with arrows emerging from sets Customer, Account, User and Session are property edges (e.g. name).

- Assertions identify invariants, which are separately defined in ADs. Assertions connected to some set are local; those standing-alone are global. In VCB, double-clicking on an assertion takes the user to its AD (symbol ⊕). In Figs. 1.2b and 1.2c, CorporateHaveNoSavings, HasCurrentBefSavings and HasSessionIffLoggedIn are global, SavingsArePositive and MaxPwMissesInv are local.

- A SD can contain constants of sets and scalars. Constants have their labels preceded by symbol ©. Scalar constants are represented as objects (rectangles); set constants as set

---

2This defines time as set of time points that are isomorphic to the natural numbers.
Figure 1.3: Sample behaviour diagrams of secure simple bank (from Amá11)

Behaviour Diagrams

A behaviour diagram (BD) declares the operations of a package. It may declare: (a) package operations to be separately specified in ADs and CDs or (b) operation compositions. Figure 1.3 presents sample BDs of secure simple bank. Specified operations of BDs are as follows:

- There are two kinds of specified operations: update (or modifiers) and observe (or queries). Queries are represented as assertions, modifiers as contracts. In Fig. 1.3b, operations UserIsLoggedIn, GetUserGivenId and IsLoggedIn are queries; all others are modifiers.

- Specified operations may be local or global: local when they are inside some set and global otherwise. The global operations of a package define the behaviour that the package offers to the outside world. Each specified operation needs to be defined in a AD or CD; double-clicking takes the user to its definition (symbol ⊔).

BDs supports three kinds of operation compositions: integral, merge and join extension. Such compositions are defined in boxes; each box is named after the kind of composition. Operation compositions are as follows:

- Integral extension promotes operations from incorporated packages to package operations so that they become available to the outside world as part of the package being defined.
The promoted operation is made available in the new package unaltered (hence name \textit{integral}). Operations to be integrally extended are represented inside an integral extension box (there is at most one); they are represented visually as normal operations connected to the package where they come from. Special operation \texttt{All} refers to all operations of a package and it may include a list of operations to exclude. BD of Fig. 1.3c says that all operations of package \texttt{TransactionsSwitch} with the exception of \texttt{IsSuspended} and \texttt{Init} are to be integrally extended; likewise for all operations of package \texttt{SecForBank} except \texttt{UserIsLoggedInAndHasPerm} and \texttt{Init}.

- Merge extension is a form of composition that merges or joins separate behaviours coming from different packages. Merge extensions are specified in a merge box; all separate operations that are included in the merge box with the same name are merged into a new package operation joining the separate behaviours. Semantically, merged operations are combined using an operator that is akin to \textit{logical conjunction}. BD of Fig. 1.3c includes a merge box saying that \texttt{Init} of \texttt{SecForBank} is to be merged with \texttt{Init} of \texttt{TransactionsSwitch}.

- Join extension is VCL’s aspect-oriented like mechanism. It inserts a certain extra behaviour into a group of operations. Join extensions involve placing the group of operations to extend inside a join box; the extra behaviours to insert are specified as \textit{join contracts}, comprising a pre- and post-condition, that is connected to the box through a \textit{fork edge}. There are two kinds of fork edges: \textit{concurrent} (symbol \(\wedge\)) and \textit{sequential} (symbol \(\otimes\)). BD of Fig. 1.3c includes a join extension, saying that all operations of \texttt{BankWithJI} are to be concurrently joined with contract \texttt{AuthACJoin}.

**Assertion Diagrams**

Assertion Diagrams (ADs) describe predicates over a single state of the modelled system. They are used to describe invariants and observe operations. Sample ADs are given in Fig. 1.4. ADs are as follows:

- An AD is made of two compartments: declarations (top) and predicate (bottom). This is illustrated in the ADs of Fig. 1.4.

- An AD may have a global or a local scope. Global ADs include names of package and assertion; local ADs include names of package, set and assertion. In Fig. 1.4, ADs of Figs. 1.4a and 1.4b are local; all others are global.

- The declarations compartment may includes variables, which are either scalar (represented as objects) or collections (represented as sets or blobs). Each variable has a name and a type. Variables can denote either inputs (name suffixed by ‘?’) or outputs (name suffixed by ‘!’). Figure 1.4d declares output set \texttt{accs!}. Figures. 1.4a, 1.4c and 1.4e declare several input and output objects.

- The declarations compartment may include imported assertions, either standing alone or combined in logical formulas. Double-clicking on an imported assertion takes the user to its AD definition (symbol \(\uparrow\)). An assertion import comprises an optional up arrow symbol \(\uparrow\), name of imported assertion with optional origin qualifier and an optional rename list. \(\uparrow\) symbol indicates that the import is total (variables and predicate are imported); when not present the import is partial (only the predicate is imported). Rename list indicates variables of imported assertion that are to be renamed (e.g. \([a]/a?\) says that \(a\) replaces \(a?\)). Fig. 1.4d has two assertion imports: one total and one partial. Total assertion import says that assertion \texttt{GetBalance} is to be called on \texttt{Account} object \(a!\); as import is total, variable
ball! defined in Account.GetBalance (Fig. 1.4a) is also defined in AccGetBalance. Partial import of GetAccountGivenAccNo means that output ball! is visible in AccGetBalance, but is not made available to the outside world; input aNo? is made available to the outside world as is also defined in AccGetBalance.

- The predicate compartment includes visual formulas combining set expressions, predicates
Figure 1.4a expresses an equality predicate using a predicate property edge to say that \( b! \) is to hold value \( \text{Account.balance} \). Figure 1.4b expresses an implication formula to say that if the property \( \text{aType} \) of \( \text{Account} \) has value \( \text{savings} \) then the property \( \text{balance} \) must be greater or equal than 0. Figure 1.4c outputs the set of accounts with negative balances: this builds a set using a set definition construction (symbol \( \bigcirc \)) by constraining set \( \text{Account} \) using predicate property edges (arrows); the constructed set is then assigned to output ‘accs!’ . Figure 1.4d expresses a set membership predicate using a predicate property edge to say that output \( a! \) belongs to accounts with property \( \text{accNo} \) equal to \( \text{aNo}\) (there is at least one). Figure 1.4f comprises a set formula that defines a set by constraining the relation \( \text{Holds} \), using property edge modifiers with operators domain (\( \bowtie \)) and range restriction (\( \lhd \)), to give the set of tuples made of corporate customers and savings accounts; outer shading (reinforced with symbol \( \exists \)) then says that this set must be empty.

**Contract Diagrams**

Contract Diagrams (CDs) describe system dynamics. They comprise a pair of predicates corresponding to pre- and post- conditions. Pre-condition describes what holds before operation is executed. Post-condition describes effect of operation, saying what holds after execution. CDs are used to describe operations that change the state of modelled system. Sample CDs are given in Fig. 1.5. CDs are as follows:

- Like ADs, CDs are made of two main compartments for declarations and predicate. In CDs, the predicate compartment is subdivided in two for pre-condition (left) and post-condition (right). This is illustrated in CDs of Fig. 1.5.

- CDs are similar to ADs in terms of what can be included in the declarations compartment. The only difference is that CDs can include both imported assertions and contracts. This is illustrated in CDs of Fig. 1.5. In Fig. 1.5a, import of contract \( \text{HasSessionAddNew} \) includes a renaming. In Fig. 1.5b, the two imported contracts are combined using a disjunction to say that a login operation is either successful (\( \text{LoginOk} \), Fig. 1.5c) or not (\( \text{LoginNotOk} \), Fig. 1.5d).

- In CDs, pre- and post- condition compartments are made of the same sort of visual formulas used in the predicate compartment of ADs. In the post-condition compartment, the variables that change state are bold-lined. In Figs. 1.5a and 1.5b, pre- and post-conditions compartments are made of arrows formulas stating equality predicates; the post-state variables are bold-lined in the post-condition compartment.

**1.1.2 Semantics**

VCL’s modelling paradigms are set theory, object-orientation and design-by-contract (pre- and post-conditions). VCL’s semantics is based on set theory. Its semantics definition takes a translational approach: diagrams are mapped to ZOO [APS05, Amá07], an object-oriented semantic domain for Z [Spi92, ISO02].

Briefly, VCL’s semantics is as follows:

- Objects are atoms; members of some set.

- Property edges are properties shared by all objects of the set. Relational edges are binary relations between sets.
• An ensemble package is defined as the conjunction of all class sets, relational edges and global invariants.

• An assertion describes a condition of a particular state structure or ensemble. It is therefore represented as a predicate over a single state structure or ensemble.

• Operations are relations between a before-state (pre-condition) and an after-state (post-condition) of particular state structure or ensemble.

Figure 1.5: Sample CDs of package Authentication of secure simple bank (from Amá11)
1.2 Outline

The remainder of this document is as follows:

• Chapter 2 presents the syntactic descriptions using metamodels and grammars. The type system is defined in the grammar.

• Chapter 3 presents the actual VCL type system made up of type rules.

• Appendix A presents the auxiliary definitions that are used to describe VCL’s type system presented here.

• Appendix B presents the VCL metamodels described using the Alloy formal modelling language.
Chapter 2
Syntax

This chapter presents the syntax of VCL package, structural behavioural and assertion diagrams. It starts by presenting the syntax of these notation using visual metamodels. Then, it presents their equivalent grammars. The next chapter defines the type system based on the grammar representation.

2.1 Metamodels

The metamodels of the VCL notations presented here have been defined in the Alloy specification language \([\text{Jac06}]\). They are given in appendix B. Here, we present these metamodels using UML class diagrams, which partially describe what is described in Alloy: the Alloy describes constraints that are not describable using class diagrams.

The Alloy metamodels for the different VCL diagram types comprise the following Alloy modules: package diagrams (section B.1), common (section B.2), structural diagrams (section B.3), common assertion and contract diagrams (section B.4), and assertion diagrams (section B.5). The following class diagrams describe each of these modules.

2.1.1 Package Diagrams

A VCL package diagram (PD) defines a package (the current package) and its relations with other VCL packages. In VCL, packages are represented using a cloud symbol. The metamodel of VCL PDs (Fig. 2.1), corresponding to the Alloy module of section B.1, is as follows:

![Figure 2.1: The metamodel of VCL Package diagrams](image-url)
• A PD (class PackageDiagram) comprises the package being defined (defines association-end), an instance of CurrPackage, which is just a special Package (inheritance relation). A CurrPackage is depicted with a bold line; it can either be Ensemble or Container (kind association-end): containers have their label preceded by symbol ⊙, and ensembles by symbol ⋆. A CurrPackage can: (a) enclose other packages to represent the packages it incorporates (inside association-end), (b) be connected with uses arrows to other packages (uses association-end) and (c) contain an extends list (ExtendsList class) to indicate sets being extended. A container cannot incorporate ensembles. All packages being incorporated and used must have been defined.

• The packages being incorporated may be connected with edges (PkgEdge). There are two kinds of edges (kind association-end): Overrides and Merges. An edge has a label indicating the blobs being overridden or merged (names attribute). Only blobs with no local properties (property edges) may be overridden. Edges define package relations that are anti-reflexive and anti-symmetric.

Figure 2.2: The common metamodel
2.1.2 Common

The metamodel of the part that is common to both SDs and ADs (Fig. 2.2), corresponding to the Alloy module of section B.2, is as follows:

- Several constructions have a name attribute; the metaclass (Name, bottom-left) denotes all names of a VCL model. Several constructions use the type designator (TypeDesignator, bottom-left). A type designator can either denote the set of natural numbers (TypeDesignatorNat), the set of integers (TypeDesignatorInt), or some set defined by a blob or relation edge and denoted by their identifier (TypeDesignatorId).

- A property edge (PropEdge) can either be of type predicate (PropEdgePred) or modifier (PropEdgeMod). PropEdgePreds comprise a unary and binary operator (uop and bop association-ends), an instance of EdgeOperatorUn and EdgeOperatorBin, respectively, a target Expression (target association-end) and an optional designator (attribute desig) to refer to some property of a blob. A PropEdgeMod comprises a modifier operator (mop association-end) an instance of EdgeOperatorMod.

- A modifier edge operator (EdgeOperatorMod) is an enumeration comprising the operators: domain restriction (DRES, △), range restriction (RRES, □), domain subtraction (DSUB, ⊖) and range subtraction (RSUB, ⊘). A predicate edge operator is enumeration comprising the operators: equality (EQ, =), non-equality (NEQ, ≠), set membership (IN, ∈), less than (LT, <), less or equal then (LEQ, ≤), greater then (GT, >), greater or equal then (GEQ, ≥), and subsetting (SubsetEQ, ⊆).

- There are two kinds of expressions: object (ObjExpression), represented as objects (rectangles), and set (SetExpression), represented as blobs (rectangles with rounded corners). An object expression can either be: an identifier (ObExpId); a number (ObjExpNum); a unary minus expression (ObjExpUMinus), comprising another expression (exp association-end); a binary object expression, comprising two expressions (association-ends exp1 and exp2) and an infix operator (bop association-end); or a parenthesised expression, comprising another expression (exp association-end). A binary object-expression operator (ObjExpBinOp) is an enumeration comprising the operators: Plus (+), Minus (−), Times (×), and Div (÷).

- A SetExpression can either refer to some existing set (SetExpressionId), denote the empty set (a blob that is shaded), be a cardinality operator applied to another set expression SetExpressionCard, or be a set definition (SetExpressionDef). A SetExpressionId comprises a designator of the set being referred (attribute desig). A SetExpressionCard is the cardinality operator applied to another set expression (sExp association-end). A SetExpressionDef comprises a set definition (association-end def), an instance of SetDef.

- Set definitions (SetDef) are defined by the things they have inside. They comprise an inside expression (insideExp association-end), representing the expression placed inside the blob, and by a set definition operator (sdop association-end). A set definition operator (SetDefOp) is an enumeration defining the operators Domain (symbol ←), Range (symbol →), Union (symbol ∪), Intersection (symbol ∩), CrossProduct (symbol ×), SetMinus (symbol \) or None (no operator).

- A set inside expression (SetInsideExpression is either an inside definition (InsideDef) or a sequence of set definitions (InsideExpSDs). A InsideExpSDs comprises a sequence of set definitions (setDefs association-end). An InsideDef is an abstract class, which comprises either a SetExtension or a ConstrainedSet. A ConstrainedSet represents
Figure 2.3: The metamodel of VCL Structural diagrams

a set constrained with an ordered collection of property edges (association-end pes). A set extension (SetExtension) represents a set defined extensionally by a set of elements (association-end elems), which are instances of SetElem.

- A SetElem is represented visually as a rectangles; it can either be a VCLObject (a member of set) or a Pair (a member of a relation). A VCLObject comprises a name (the name of the object); a Pair comprises a pair of names.

2.1.3 Structural Diagrams

The metamodel of VCL structural diagrams (SDs) (Fig. 2.3), corresponding to the Alloy module of section B.3, is as follows:

- A SD (SDDiag) is made of structural elements (SDElem) and invariants (Assertion). A SDElem can be a relation edge (RelEdge), constant (Constant) or set (Set).

- In a SD, an Assertion represents an invariant. If they belong to the overall SD (association-end invariants) they represent global invariants; if they are connected to a set (association-end lInvariants), the invariant is local to the set.

- A relation edge (RelEdge), or association, represents an edge between two sets: the source and the target. It holds two attributes to record the multiplicities attached to source and target (multS and multT).

- Like invariants, constants (Constant) are global if they are not connected to any set and local otherwise (association-end lConstants).
• A set can be primary (PrimarySet), derived (DerivedSet) or one of the sets corresponding to primitive types: integers (IntSet) or natural numbers (NatSet).

• A derived set has a name (attribute id) and is associated with a set definition (SetDef, defined in common metamodel).

• A primary set has a kind (SetKind), indicating whether the set is Class or Value. A primary set comprises a set of local constants (association-end lConstants), a set of local invariants (association-end lInvariants), and a set of property edge definitions (association-end lProps). A primary set may have other primary sets and objects inside (association-ends hasInsideSet and hasInsideO).

• A property edge definition (PropEdgeDef) has a set has the edge’s target (association-end peTarget) indicating the type of the property, and a multiplicity constraint (attribute mult).

• In SDs, VCL objects (VCLObject) may be placed inside blobs to represent objects of some set. This construction is used to define enumerations in VCL SDs.

• Multiplicities (Mult) are attached to relation edges and property edge definitions. A multiplicity can either be single (MOne), optional (MOpt), sequence (MSeq), multiple with 0 or more (Many), multiple with at least one (MOneToMany), or be defined as a range (MRange) comprising a lower and an upper bound (association-ends ub and lb).

2.1.4 Common Assertion and Contract Diagrams

The metamodel of common assertion and contract diagrams (Fig. 2.4), corresponding to the Alloy module of section B.4, is as follows:
• A declarations compartment (DeclCompartment) comprises several declarations (Decl), which can either be a typed declaration (TypedDecl) or a declaration formula (DeclFormula). A typed declaration has a name (dName) and a type (dTy), and it can either be a declaration of an object (DeclObj) or the declaration of a set (DeclSet). The sequence attribute of DeclSet indicates whether the set is a normal set (value false) or a sequence (value true). The optional attribute of DeclObj indicates whether the object is optional or not.

• A declarations formula (DeclFormula) can either be a declarations formula atom (DeclFormulaAtom), which comprises a declaration reference, a negated declaration formula (DeclFormulaNot), which comprises the declarations formula being negated, or a binary declaration formula (DeclFormulaBin), which comprises an operator (DeclFormulaBinOp) and two declarations formulas.

• FormulaSource represents the source of a predicate formula in AD or CDs. It can either be a formula source element (FormulaSourceElem), which comprises a set Element (defined in Common, Fig. 2.2), or a blob element (FormulaSourceBlob).

2.1.5 Assertion Diagrams

The metamodel of VCL assertion diagrams (Fig. 2.5), corresponding to the Alloy module of section B.5, is as follows:

• An assertion diagram (ADiag) comprises a name (aName), a set of declarations corresponding to the declarations compartment (declarations association-end), and a set of formulas corresponding to the predicate compartment (predicate association-end).

• A formula (Formula) can either be a negation formula (FormulaNot), a binary formula (FormulaBin), an object formula (ObjFormula) or a set formula (SetFormula).

• A negation formula (FormulaNot) comprises another formula corresponding to the formula being negated (frml association-end). A binary formula (FormulaBin) comprises two formulas corresponding to the formulas being combined (frml1 and frml2 association-ends),
VCL ::= \overline{Pkg}

Pkg ::= PD SD AD

Figure 2.6: Syntax of VCL Models

and an operator (op association-end). A binary formula operator (FormulaBinOp) can either be an implication (implies), a conjunction (and), a disjunction (or), or an equivalence (equiv).

• An arrows formula (ArrowsFormula) comprises a set of predicate property edges (pes association-end).

• A set formula (SetFormula) can either be a subset formula (FormulaSubset), a shaded blob formula (SetFormulaShaded) or a set definition formula (SetFormulaDef). A subset formula (FormulaSubset) corresponds to the situation where one set is placed inside another to denote the subset relationship; it has a set identifier (attribute setId) and a set expression to denote the inside set (hasInside association-end). A shaded set formula corresponds to the situation where some set is shaded; it comprises a set identifier (attribute setId). A definition set formula (SetFormulaDef) comprises a SetDef (association-end setdef) from the common metamodel (Fig. 2.2); it can be shaded or have an identifier (either one or the other).

2.2 Grammars

The metamodels presented above are the basis for the implementation of diagram editors in VCL’s tool. To specify type systems, grammars are a more convenient representation. The following presents the grammars of VCL package, structural and assertion diagrams; they are equivalent to the visual metamodels presented above.

The grammars use the following operators:

• \(x\) for zero or more repetitions of x;
• \(x^1\) for one or more repetitions of x;
• \(x|y\) for a choice of x or y;
• \([x]\) for an optional x.

In addition,

• \(xc\) for some character symbol \(c\) means zero or more occurrences of x separated with \(c\);
• \(xc^1\) for some character symbol \(c\) means one or more occurrences of x separated with \(c\);

Symbols are set in bold type when they are to be interpreted as terminals to avoid confusion with grammar symbols. We introduce two syntactic sets, representing terminals: the set of identifiers \(Id\), and the set of numeric constants \(Num\).

Figure 2.6 defines the grammar of a VCL model. A VCL model (VCL) comprises a sequence of package models (Pkg). A package model (Pkg) comprises one package diagram (PD), one structural diagram (SD) and several assertion diagrams (ADS).

The grammars defining the syntax of the different VCL diagrams are as follows:
• Figure 2.7 presents the grammar of package diagrams (PDs).
• Figure 2.8 describes the syntactic constructions that are common to both SDs and ADs.
• Figure 2.9 presents the grammar of structural diagrams (SDs).
• Figure 2.10 presents the grammar of assertion diagrams (ADs).

PK ::= container | ensemble
PD ::= PK package Id [uses PRef,][incorporates PRef] { PDep [PExts] }
PRef ::= Id [as Id]
PExts ::= extends (Td,)
PDep ::= Id PEdgKind Id on (Td)
PEdgKind ::= merges | overrides

Figure 2.7: Syntax of Package Diagrams

TD ::= int | nat | [ld ::] ld
O ::= object ld
P ::= pair (ld, ld)
SE ::= O | P
A ::= assertion Ald
PE ::= (PEP | PEM) TExp
PEP ::= [UEOp] [ld.] ! BEOp
UEOp ::= # j ◇ j ▷
BEOp ::= = | ̸= j < j ≤ j > j ≥ j ⊂
PEN ::= [SOp ]
TExp ::= FExp | SExp
FExp ::= [ld ::] ld | ld.Id | Num | ¬FExp | FExp FEOP FExp | (FExp)
FEOP ::= + | − | * | div
SExp ::= set TD | SDef | set shaded | # SExp
SDef ::= set ⊔ SOp hasIn {IExp}
SOp ::= ◐ | → | ∩ | ∪ | × | \ | < | > | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ⊊ | ⊊ | ⊆ | ⊇ | ⊂ | ⊃ | ATEGORY 2.8: Common syntactic constructions to ADs and SDs

20
Figure 2.9: Syntax of structural diagrams

AD ::= AD Id [ ld ] decls { D } pred { F }  
D ::= VD | DF  
VD ::= [ hidden ] DV TR : TD  
DV ::= [ opt ] object | set | seq  
R ::= Id / Id  
DFA ::= [ ] A [ TR ] | [ ] assertion Id ( → | . | :: ) Id [ TR ]  
DF ::= DFA | FOP[ DF ]  
FOP ::= ⇒ | ⇐ | ∧ | ∨ | ⊑ | ⊑  
F ::= AF | SF | FOP[ F ] | QF  
AFS ::= SE | AFSS | FSOP AFS  
AFSS ::= set Id | SDef  
FSOP ::= # | ← | → | *  
AF ::= AFS { PED }  
SF ::= [ shaded ] Id SDef | set shaded TD | set TD hasIn { SExp }  
QF ::= QDF ; • F ;  
QD ::= ( ∀ | ∃ ) TDD ;

Figure 2.10: Syntax of Assertion Diagrams
Chapter 3
Type System

This chapter presents VCL’s type system. It starts by defining VCL’s types and typing environments (section 3.1). Then, it presents the basic (section 3.2) and common rules (section 3.3) of the type system. Finally, it presents the rules that are specific to package (section 3.4), structural (section 3.5) and assertion diagrams (section 3.6).

3.1 Types and Environments

A variable environment (VE) denotes a set of bindings, mapping identifiers to their types:

\[ VE == Id \rightarrow T \]

VCL’s types (set T) are as follows:

\[ T ::= \text{Int} | \text{Nat} | \text{Null} | \text{Pow} T | \text{Seq} T | \text{Opt} T | \text{Top} | \text{Obj} | \text{Set} \text{Id}::\text{Id} | \text{Pair} (T, T) \]

Here, (a) Int represents the integers, (b) Nat the natural numbers; (c) Null represents erroneous results (implementation only); (d) Pow T represents a powerset of some set; (e) Seq T represents a sequence of some type; (f) Opt T represents an optional (either it exists or is empty); (g) Top is a maximal type (type of all well-formed terms); (h) Obj is the maximal type of all well-formed objects; (i) Set represents primary sets, comprising two identifiers to represent the set’s package and the set’s name; (j) Pair represents a cartesian product of two types; (k) Assertion represents assertions (variable environments indicate assertion’s variables, which are either visible, VE_v, or hidden, VE_h); (l) Contract represents assertions; and (i) Pkg represents packages.

VCL’s type rules use and manipulate environments (set E below), which are made of four components: (a) variable, (b) package, (c) set and (d) subtyping. Variable environments give the type bindings of some scope. Package environments (PE) map identifiers to a pair made up of the package’s kind (PK, Fig. 2.7) and the package’s environment. Set environments (SE) map identifiers to a quadruple made up of the set’s kind (value or class), definitional status (DK), identifier of originating package and local variable environment. Subtyping environments (set SubE) are the subtyping relations between types:
Table 3.1 Judgements associated with the basic rules of VCL’s type system

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash T$</td>
<td>$T$ is well-formed type in $E$</td>
</tr>
<tr>
<td>$E \vdash T_1 &lt;: T_2$</td>
<td>$T_1$ is a subtype of $T_2$ in $E$</td>
</tr>
<tr>
<td>$E \vdash Id : T$</td>
<td>$Id$ is well-formed identifier of type $T$ in $E$</td>
</tr>
<tr>
<td>$E \vdash Id_s : Id : T$</td>
<td>$Id_s$ is well-formed identifier of set $Id_s$ with type $T$ in $E$</td>
</tr>
<tr>
<td>$E \vdash Id_p : Id : T$</td>
<td>$Id_p$ is well-formed identifier of package $Id_p$ with type $T$ in $E$</td>
</tr>
</tbody>
</table>

$SK ::= value \mid class$

$DK ::= def \mid notDef$

$PE ::= Id \rightarrow PK \times E$

$SE ::= Id \rightarrow SK \times DK \times Id \times VE$

$SubE ::= T \leftrightarrow T$

$E ::= VE \times PE \times SE \times SubE$

A VCL diagram environment ($DE$) maps package identifiers ($Id$) to package definitions ($Pkg$, Fig. 2.6), made of one PD, one SD and several ADs. This is used to retrieve package definitions during type-checking.

$$DE ::= Id \rightarrow Pkg$$

We introduce the following conventions:

- $X$ is a sequence of some set $X$.
- $E_\emptyset$ is an empty environment. $E_.VE$, $E_.PE$, $E_.SE$ and $E_.SubE$ denote the different components of $E$.
- $Id : T$, $Id \mapsto (SK, DK, Id, VE)$ and $Id \mapsto (PK, E)$ are type ($VE$), set ($SE$) and package ($PE$) bindings. $T_1 <: T_2$ denotes a subtyping tuple, saying that $T_1$ is a subtype of $T_2$.
- Two disjoint environments are combined using $E_1 \uplus E_1$. Bindings are added to an environment using $E$, $Id : T$; similarly for other types of bindings. $E \oplus VE$ means that the environment $E$ is overridden with the set of type bindings $VE$; similarly for other types of bindings. These operators are defined precisely in appendix A.1.

3.2 Basic Rules

The basic type rules of VCL’s type system manipulate environments and define subtype relations. The judgements are listed in table 3.1. The first judgement asserts that the type $T$ is well-formed in the environment $E$. The second judgement asserts that the type $T_1$ is a subtype of $T_2$ in the environment $E$. The third judgement says that $Id$ is a well-formed identifier with type $T$ in $E$. The fourth judgement asserts the well-formedness of a set-property access; it says that property $Id_l$ of set named $Id_s$ has type $T$ in $E$.

Table 3.2 lists basic rules concerning types. Rule Type says that some identifier yields type $T$ provided the variable binding is defined in the variable environment ($E_.VE$). Rule Type describes the conditions for some type to be valid in some environment $E$: set and package types.
Table 3.2 Basic VCL typing rules

<table>
<thead>
<tr>
<th>SubTy</th>
<th>Sub Refl</th>
<th>Sub Trans</th>
<th>Subumption</th>
<th>Sub Top</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash T_1$ $E \vdash T_2$</td>
<td>$E \vdash T$</td>
<td>$E \vdash T_A : T_B$</td>
<td>$E \vdash T_A : T_B$</td>
<td>$E \vdash T$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sub Obj</th>
<th>Sub NatInt</th>
<th>Sub Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash \text{Set Id}_d : \text{Obj}$</td>
<td>$E \vdash \text{Nat} : \text{Int}$</td>
<td>$E \vdash \text{Pair}(T_{A1}, T_{B1}) : \text{Pair}(T_{A2}, T_{B2})$</td>
</tr>
<tr>
<td>$E \vdash \text{Opt} T_A : \text{Opt} T_B$</td>
<td>$E \vdash \text{Opt} T_A : \text{Pow} T_B$</td>
<td>$E \vdash \text{Opt} T_A : \text{Pow} T_B$</td>
</tr>
</tbody>
</table>

Table 3.2 lists basic subtyping rules. Rule $\text{SubTy}$ checks whether some type is a subtype of another; this amounts to check that both types are defined and that the subtyping tuple belongs to the environment’s set of subtypes $(E, \text{SubE})$. Rules $\text{Sub Refl}$ and $\text{Sub Trans}$ says that the subtyping relation is both reflexive and transitive. Rule $\text{Subumption}$ is the subsumption rule that says that if some variable has type $T_A$, and if $T_A$ is subtype of $T_B$ then the variable also has type $T_B$. The remaining rules define subtype relations between types. Rule $\text{Sub Top}$ says that any valid type is a subtype of type $\text{Top}$. Rule $\text{Sub Obj}$ says that any set type is a subtype of type $\text{Obj}$. Rule $\text{Sub NatInt}$ says that type of natural numbers is a subtype of the integers. Rules $\text{Sub Pow}$, $\text{Sub Seq}$ and $\text{Sub Opt}$ say, respectively, that two powerset, sequence or optional types are subtypes of each other provided their enclosed types ($T_A$ and $T_B$) are also. Rule $\text{Sub Opt PSet}$ says that optional types are a subtype of powerset types provided their enclosed types ($T_A$ and $T_B$) are also. Finally, rule $\text{Sub Pair}$ says that two pair types are subtypes of each other if their corresponding components are subtypes of each other also.

are valid provided their identifiers are defined in set and package environments; all remaining types are valid. Rule $\text{Set Id Prop}$ yields the type that is associated with some local identifier $\text{Id}_d$ of some set $\text{Id}_d$; the rule checks that the set type is defined and then retrieves the type of the local identifier from the set’s variable environment. Rule $\text{Package Id}$ retrieves the type of some identifier $\text{Id}$ defined in some package $\text{Id}_d$; this requires that there is a package type defined for the given package identifier $\text{Id}_d$ and that the given identifier is defined in the package’s environment.

Basic VCL sub-typing rules

<table>
<thead>
<tr>
<th>(Type)</th>
<th>(SetId Prop)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T = \text{Set Id}_d : \text{Id}_d$ $\Rightarrow \text{Id}_d \in \text{dom E.SE}$</td>
<td>$E \vdash \text{Set Id}_d : \text{Id}_d$ $\text{Id}_d \in \text{dom}(E, \text{SE}(\text{Id}_d))$ $\text{VE}$</td>
</tr>
<tr>
<td>$T = \text{Pkg Id} \Rightarrow \text{Id} \in \text{dom E.PE}$</td>
<td>$E \vdash \text{Pkg Id} : T$</td>
</tr>
</tbody>
</table>

$E \vdash \text{Id}_d : \text{Id}_d : T$
presents rules for checking the well-formedness of assertions (2.8 AD 3.7 E SDef AD 3.6 E). An inside definition (and then apply the rule for the set definition’s operator (essentially derive a sequence of types from inside definition (Pair rule builds a pair type from the types of its constituent identifiers. has already been type-checked and its information can, therefore, be retrieved from the environment. The assertion rule assumes that the AD associated with the assertion being checked (some foreign package being used (rule Nat well-formedness of different terms of the grammar that is common to SDs and ADs of Fig. 3.3 Common Rules (Table 3.5) Type rules for type designators (TD non-terminal)

<table>
<thead>
<tr>
<th>(TD Nat)</th>
<th>(TD Int)</th>
<th>(TD Id)</th>
<th>(TD Id Pkg)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E\vdash Id : T</td>
<td>E\vdash Id : T</td>
<td>E\vdash Id : T</td>
<td>E\vdash Id : T</td>
</tr>
<tr>
<td>E\vdash Id : T</td>
<td>E\vdash Id : T</td>
<td>E\vdash Id : T</td>
<td>E\vdash Id : T</td>
</tr>
</tbody>
</table>

3.3 Common Rules

The judgements for the common part of the VCL type system are given in table 3.4. They assert the well-formedness of different terms of the grammar that is common to SDs and ADs of Fig. 2.8 (chapter 2). Type designator rules (Table 3.4) derive a type from a designator, yielding a primitive type (Int or Nat) or some type that is associated with an identifier either from the current package (rule TD Id) or some foreign package being used (rule TD Id Pkg). Table 3.4 presents rules for checking the well-formedness of assertions (T Assertion), VCL objects (T SE Obj) and pairs (T SE Pair). These rules merely extracts the types associated with identifiers from the environment. The assertion rule assumes that the AD associated with the assertion being checked has already been type-checked and its information can, therefore, be retrieved from the environment. Pair rule builds a pair type from the types of its constituent identifiers.

A set definition (SDef nonterminal, Fig. 2.8) is a syntactic construct to build sets. The type rules for set definitions (table 3.7) consider two cases, depending on whether the inside expression comprises one inside definition (rule T SDef IDef) or a sequence of set definitions (rule T SDef SDef) and then apply the rule for the set definition’s operator (SOp) to retrieve the types yielded by the rules. An inside definition (IDef nonterminal, Fig. 2.8) is a construction associated with set definitions. An inside definition can either be a constrained set or a set expression. The type rules for inside definitions (table 3.7) consider these two cases. The constrained set rule (IDef CntSet) derives a type from the given type designator (TD) and then checks the sequence of property edges in the context of this derived type (T); the rule says that the set of property edges must either be of only one kind: either predicate or modifier (disjunction). The type rules for set extensions (IDef SE and IDef SE *) process the sequence
Table 3.6 Type rules for assertions and set elements

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (T \text{ Assert}) )</td>
<td>( E \vdash Id : T )</td>
</tr>
<tr>
<td>( T = \text{Assertion}[VE_v, VE_h] )</td>
<td>( E \vdash Id : T )</td>
</tr>
<tr>
<td>( (T SE \text{ Obj}) )</td>
<td>( T \text{ &lt;; Obj} )</td>
</tr>
<tr>
<td>( (T SE \text{ Pair}) )</td>
<td>( E \vdash \text{pair}(Id_1, Id_2) : \text{Pair}(T_1, T_2) )</td>
</tr>
<tr>
<td>( E \vdash Id : T )</td>
<td>( E \vdash Id : T )</td>
</tr>
<tr>
<td>( E \vdash Id : T )</td>
<td>( E \vdash Id : T )</td>
</tr>
</tbody>
</table>

Table 3.7 Type rules for set definitions and associated inside definitions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (T S\text{Def }I\text{Def}) )</td>
<td>( E \vdash i^{\text{sd}} \text{Def} : T_1 )</td>
</tr>
<tr>
<td>( E ; S\text{Op}^{\text{sd}o} T_1 : T )</td>
<td>( E ; S\text{Op}^{\text{sd}o} T_1 : T )</td>
</tr>
<tr>
<td>( \text{set } \bigcirc \text{ SOp hasIn } {I\text{Def}} : T )</td>
<td>( E ; S\text{Op}^{\text{sd}o} T_1 : T )</td>
</tr>
<tr>
<td>( (I\text{Def }\text{CnSet}) )</td>
<td>( (I\text{Def }\text{SE }s) )</td>
</tr>
<tr>
<td>( E \vdash i^{\text{sd}} \text{Def} )</td>
<td>( E \vdash i^{\text{sd}} \text{Def} )</td>
</tr>
<tr>
<td>( E ; T \vdash ^{\text{PE}} \text{PE} : T' )</td>
<td>( E \vdash ^{\text{PE}} \text{PE} : T' )</td>
</tr>
<tr>
<td>( (\text{IsPEP}(\text{PE}) \lor \text{IsPEM}(\text{PE})) )</td>
<td>( E \vdash ^{\text{PE}} \text{PE} : T' )</td>
</tr>
<tr>
<td>( E \vdash i^{\text{sd}} \text{Def} )</td>
<td>( E \vdash i^{\text{sd}} \text{Def} )</td>
</tr>
<tr>
<td>( (\text{IsPEP}(\text{PE}) \lor \text{IsPEM}(\text{PE})) )</td>
<td>( E \vdash ^{\text{PE}} \text{PE} : T' )</td>
</tr>
<tr>
<td>( E \vdash i^{\text{sd}} \text{Def} )</td>
<td>( E \vdash i^{\text{sd}} \text{Def} )</td>
</tr>
<tr>
<td>( (\text{IsPEP}(\text{PE}) \lor \text{IsPEM}(\text{PE})) )</td>
<td>( E \vdash ^{\text{PE}} \text{PE} : T' )</td>
</tr>
</tbody>
</table>

of set elements inductively; retrieving the greatest type of all the elements in the sequence, which must be subtypes of each other.

The rules for set definition operators (\(\text{SOp}\) non-terminal) apply to a sequence of types in the context of an environment and a set definition operator; they are given in table 3.8. The rules are as follows:

- Rule \(\text{SOp None}\) considers the case where there is no operator. The rule requires that the sequence of types is made of a single element, and yields the type given in the sequence.
- Rules for domain and range operators (\(\text{SOp Dom}\) and \(\text{SOp Ran}\)) require that there is a single type given in the sequence and that this type is a powerset of a pair (it is a binary relation). Rule \(\text{SOp Dom}\) returns a type formed as the powerset of the first type of the pair (the domain). Rule \(\text{SOp Ran}\) returns a type formed as the powerset of the second type (the range).
- The cross product rules (\(\text{SOp Cross}\) and \(\text{SOp } \ast \text{Cross}\)) consider two cases depending on whether the sequence is made of a pair of types or more than a pair. The pair rule takes a pair of power sets and yields a powerset of a pair type. Rule \(\text{SOp } \ast \text{Cross}\) takes a powerset type and a sequence of types and returns a powerset of a pair type formed with the derived type.
- The intersection (\(\text{SOp Pair Intersection}\) and \(\text{SOp } \ast \text{Intersection}\)) and union rules (\(\text{SOp Pair Union}\) and \(\text{SOp } \ast \text{Union}\)) take a sequence of at least two powerset types and return a powerset of the greatest type in the sequence, according to the subtyping relation (function \(\text{getGType}\), appendix A). All given types must be subtypes of each other. The set subtraction rule (\(\text{SOp Pair SetMinus}\)) does the same for a pair of powerset types.
Table 3.8 Type rules for set def operators

<table>
<thead>
<tr>
<th>(SOp None)</th>
<th>(SOp Dom)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E; \bot \vdash \text{Pow} ) ( : \text{T} )</td>
<td>( E; \vdash \text{Pow} ) ( T_d, T_r ) ( : \text{Pow} ) ( T_d )</td>
</tr>
<tr>
<td>(SOp Ran)</td>
<td></td>
</tr>
</tbody>
</table>

\[
\begin{align*}
E; \vdash \text{Pow Pair} \ (T_r, T_r) & : \text{Pow} \ T_r \\
E; \times \vdash \text{Pow Pair} \ (T_1, T_2) & : \text{Pow} \ (T_1, T_2) \\
E; \cap \vdash \text{Pow Pair} \ (T_1, T_2) & : \text{Pow} \ T_1 \\
E; \cup \vdash \text{Pow Pair} \ (T_1, T_2) & : \text{Pow} \ T_1 \\
E; \mid \vdash \text{Pow Pair} \ (T_1, T_2) & : \text{Pow} \ T_1
\end{align*}
\]

Table 3.9 Type rules for property edges

<table>
<thead>
<tr>
<th>(PE PEP)</th>
<th>(PE PEM)</th>
<th>(PE *)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E; \vdash \text{PEP} ) ( T_\text{Exp} : T_r )</td>
<td>( E; \vdash \text{PEM} ) ( T_\text{Exp} : T_r )</td>
<td>( E; \vdash \text{PE} ) ( T_r )</td>
</tr>
<tr>
<td>( E; (T, T_1) \vdash \text{PEP} ) ( T_\text{Exp} : T_r )</td>
<td>( E; (T, T_1) \vdash \text{PEM} ) ( T_\text{Exp} : T_r )</td>
<td>( E; T_r \vdash \text{PE} ) ( T_r )</td>
</tr>
<tr>
<td>(PEP)</td>
<td>(PEPS)</td>
<td>(PEPM)</td>
</tr>
<tr>
<td>( E; T_r \vdash \text{PEP} ) ( \mid \vdash \text{Id} : T_r ) ( T_r' )</td>
<td></td>
<td>( E; T_r \vdash \text{PEM} ) ( \mid \vdash \text{Id}_{\text{PrId}} : T_r ) ( T_r' )</td>
</tr>
<tr>
<td>( E; \vdash \text{UEOp} \ (T_\text{Exp} : T_r) ) ( T_r' )</td>
<td>( E; \vdash \text{UEOp} \ (T_\text{Exp} : T_r) ) ( T_r' )</td>
<td>( E; \vdash \text{MOOp} \ (T, T_r) ) ( T_r' )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(UEOp No)</th>
<th>(UEOp Card)</th>
<th>(UEOp The)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E; (T, T_1) \vdash \text{UEOp} ) ( \mid \vdash \text{Id} \rightarrow \text{BEOP} )</td>
<td>( E; \vdash \text{UEOp} ) ( \mid \vdash \text{Id}_{\text{PrId}} : T_r ) ( T_r' )</td>
<td>( E; (T, T_1) \vdash \text{MOOp} ) ( \mid \vdash \text{MOOp} ) ( T_r' )</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
E; \vdash \text{UEOp} \ (T) & : \text{T} \\
E; \vdash \text{UEOp} \ (\text{Pow} \ T) & : \text{Int} \\
E; \vdash \text{UEOp} \ (\text{Opt} \ T) & : \text{T}
\end{align*}
\]
Table 3.10 Type rules for binary predicate edge operators (BEOp)

<table>
<thead>
<tr>
<th>Type Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{BEOp \ EQ}) )</td>
<td>( E \vdash T_1 &lt;: T_2 \lor E \vdash T_2 &lt;: T_1 )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(T_1, T_2) )</td>
<td>( E; \vdash \mathcal{E'}(T_1, T_2) )</td>
</tr>
<tr>
<td>( (\text{BEOp \ LT}) )</td>
<td>( E \vdash T_1 &lt;: \text{Int} )  ( E \vdash T_2 &lt;: \text{Int} )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(T_1, T_2) )</td>
<td>( E; \vdash \mathcal{E'}(T_1, T_2) )</td>
</tr>
<tr>
<td>( (\text{BEOp \ GT}) )</td>
<td>( E \vdash T_1 &lt;: \text{Int} )  ( E \vdash T_2 &lt;: \text{Int} )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(T_1, T_2) )</td>
<td>( E; \vdash \mathcal{E'}(T_1, T_2) )</td>
</tr>
<tr>
<td>( (\text{BEOp \ SUBSETEQ}) )</td>
<td>( E \vdash T_1 &lt;: T_2 )</td>
</tr>
<tr>
<td>( E; \subseteq \mathcal{E'}(\text{Pow} \ T_1, \text{Pow} \ T_2) )</td>
<td>( E; \subseteq \mathcal{E'}(\text{Pow} \ T_1, \text{Pow} \ T_2) )</td>
</tr>
</tbody>
</table>

Table 3.11 Type rules for modifier edge operators (MOp)

<table>
<thead>
<tr>
<th>Type Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( (\text{MOp \ DRES}) )</td>
<td>( E \vdash T &lt;: T_d )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(\text{Pow Pair} \ (T_d, T_r), \text{Pow} \ T) : \text{Pow Pair} \ (T_d, T_r) )</td>
<td>( (\text{MOp \ RRES}) )</td>
</tr>
<tr>
<td>( E \vdash T &lt;: T_r )</td>
<td>( E \vdash T &lt;: T_r )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(\text{Pow Pair} \ (T_d, T_r), \text{Pow} \ T) : \text{Pow Pair} \ (T_d, T_r) )</td>
<td>( (\text{MOp \ DSUB}) )</td>
</tr>
<tr>
<td>( E \vdash T &lt;: T_r )</td>
<td>( E \vdash T &lt;: T_r )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(\text{Pow Pair} \ (T_d, T_r), \text{Pow} \ T) : \text{Pow Pair} \ (T_d, T_r) )</td>
<td>( (\text{MOp \ RSM}) )</td>
</tr>
<tr>
<td>( E \vdash T &lt;: T_r )</td>
<td>( E \vdash T &lt;: T_r )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(\text{Pow Pair} \ (T_d, T_r), \text{Pow} \ T) : \text{Pow Pair} \ (T_d, T_r) )</td>
<td>( (\text{MOp \ INTERSECT}) )</td>
</tr>
<tr>
<td>( E \vdash T &lt;: T_d )</td>
<td>( E \vdash T &lt;: T_d )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(\text{Pow Pair} \ (T_d, T_r), \text{Pow} \ T) : \text{Pow} \ T )</td>
<td>( (\text{MOp \ SETMINUS}) )</td>
</tr>
<tr>
<td>( E \vdash T &lt;: T_d )</td>
<td>( E \vdash T &lt;: T_d )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(\text{Pow Pair} \ (T_d, T_r), \text{Pow} \ T) : \text{Pow} \ T )</td>
<td>( E \vdash \mathcal{E'}(\text{Pow} \ T_1, \text{Pow} \ T_2) : \text{Pow} \ T )</td>
</tr>
<tr>
<td>( E \vdash T &lt;: T_d )</td>
<td>( E \vdash T &lt;: T_d )</td>
</tr>
<tr>
<td>( E; \vdash \mathcal{E'}(\text{Pow Pair} \ (T_d, T_r), \text{Pow} \ T) : \text{Pow} \ T )</td>
<td>( E \vdash \mathcal{E'}(\text{Pow} \ T_1, \text{Pow} \ T_2) : \text{Pow} \ T )</td>
</tr>
</tbody>
</table>

28


<table>
<thead>
<tr>
<th>Type rule</th>
<th>Table 3.12 Type rules for set expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\text{SExp TD})$</td>
<td>$E \vdash^{TD} TD : T$</td>
</tr>
<tr>
<td>$E \vdash^{TD} \text{set TD} : \text{Pow} T$</td>
<td>$(\text{SExp SDef})$</td>
</tr>
<tr>
<td>$E \vdash^{SDef} \text{set shaded} : \text{Pow Top}$</td>
<td>$(\text{SExp Empty})$</td>
</tr>
<tr>
<td>$(\text{SExp Card})$</td>
<td>$E \vdash^{#} \text{SExp} : \text{Int}$</td>
</tr>
</tbody>
</table>

29
Table 3.13 Type rules for free expressions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(FExp\ ID)$</td>
<td>$E\vdash E\vdash Id : T$</td>
<td>$E\vdash Id : T$</td>
</tr>
<tr>
<td>$(FExp\ ID\ Pkg)$</td>
<td>$E\vdash E\vdash Id_p::Id : T$</td>
<td>$E\vdash Id_p::Id : T$</td>
</tr>
<tr>
<td>$(FExp\ Dot)$</td>
<td>$E\vdash Id_p : \Set E\vdash Id_p::Id : T$</td>
<td>$E\vdash Id : T$</td>
</tr>
<tr>
<td>$(FExp\ Num)$</td>
<td>$E\vdash E\vdash Num : Nat$</td>
<td>$E\vdash E\vdash FE : Int$</td>
</tr>
<tr>
<td>$(FExp\ Uminus)$</td>
<td>$E\vdash E\vdash FE : Int$</td>
<td>$E\vdash E\vdash FE : Int$</td>
</tr>
<tr>
<td>$(FExp\ FEOP)$</td>
<td>$E\vdash E\vdash FE1 : Int$</td>
<td>$E\vdash E\vdash FE1 : Int$</td>
</tr>
</tbody>
</table>

Table 3.14 Type rules for checking assertions

<table>
<thead>
<tr>
<th>Rule</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Assertion Ok)</td>
<td>$Id_{FA} = \text{getFAId}(Id_{A\perp}; Id_A)$</td>
<td>$\widehat{AD} = \text{getDepsOfAD}(AD; \overrightarrow{AD}; Id_{A\perp})$</td>
</tr>
<tr>
<td></td>
<td>$AD = \text{findAD}(\overrightarrow{AD}; Id_{FA}; Id_{A\perp})$</td>
<td>$E; \overrightarrow{AD}; Id_{A\perp}; Id_{A\perp} \vdash \overrightarrow{AD} : VE$</td>
</tr>
<tr>
<td></td>
<td>$E; \overrightarrow{AD}; Id_{A\perp}; Id_{A\perp} \vdash \overrightarrow{AD} : VE$</td>
<td>$Id_{A\perp} \not\perp \Rightarrow \overrightarrow{AD} = {}$</td>
</tr>
<tr>
<td></td>
<td>$E; \overrightarrow{AD}; Id_{A\perp}; Id_{A\perp} \vdash \overrightarrow{AD} : VE$</td>
<td>$E; \overrightarrow{AD}; Id_{A\perp}; Id_{A\perp} \vdash \overrightarrow{AD} : VE$</td>
</tr>
<tr>
<td>(AD Ok)</td>
<td>$\widehat{AD} = \text{getDepsOfAD}(AD; \overrightarrow{AD}; Id_{A\perp})$</td>
<td>$E; \overrightarrow{AD}; Id_{A\perp}; Id_{A\perp} \vdash \overrightarrow{AD} : VE$</td>
</tr>
<tr>
<td></td>
<td>$E; \overrightarrow{AD}; Id_{A\perp}; Id_{A\perp} \vdash \overrightarrow{AD} : VE$</td>
<td>$E; \overrightarrow{AD}; Id_{A\perp}; Id_{A\perp} \vdash \overrightarrow{AD} : VE'$</td>
</tr>
</tbody>
</table>

Table 3.14 presents the rules for checking ADs associated with some assertion. These rules are used when the AD type information is to be loaded into the environment. The rules are as follows:

- Rule **Assertion Ok** derives the name of the assertion diagram through function \text{getFAId}, which considers the special case of assertions associated with constants, and then looks for the AD using function \text{findAD} (both functions defined in appendix A, section A.3.9). The retrieved AD is then checked (rule associated with judgement $\vdash \overrightarrow{adok}$) to yield variable environment $VE$.

- Rule **AD Ok** processes a single AD. It retrieves all the ADs that are included in the given AD through function \text{getDepsOfAD} (appendix A, section A.3.8) to yield set $\overrightarrow{AD}$ and then checks them using the rules associated with judgement $\vdash \overrightarrow{adok}$ to yield variable environment $VE$. The current AD is also checked using the rule for assertion diagrams to yield a variable binding. The rule yields a variable environment formed by adding the retrieved variable binding to the variable environment $VE$.

- Rules **AD Ok $\emptyset$** and **AD Ok $*$** process a set of ADs inductively. Rule **AD Ok $\emptyset$** considers the case where the set is empty, yielding an empty set of variable bindings. Rule **AD Ok $*$** considers the case where the set has at least one element; it builds a variable environment by joining the variable environment derived from the current single AD and the variable environment derived from the remaining set of ADs.
3.4 Package Diagrams

The judgements for package diagrams are listed in Table 3.15. The first judgement says that the package diagram (PD) is well-formed in the diagram environment $DE$, yielding a triple $(Id_p, PK, E)$, comprising the package’s identifier and kind, and an initial environment for the current package resulting from processing the PD. The remaining judgements assert the well-formedness of the different components of a PD. This comprises package uses (judgement labelled $\text{PUses}$), package dependencies ($\text{PDep}$), package merges ($\text{PMerge}$), package overrides ($\text{POver}$) and package incorporations ($\text{PInc}$).

The type rule for PDs (rule $\text{Ok PD}$, Table 3.16) checks: (a) that there are no name clashes in imports and incorporates using predicate $\text{NoClashes}$ (appendix A, section A.2.3), (b) the well-formedness of packages being used ($\text{PRef}_1$) and incorporated ($\text{PRef}_2$) to produce an updated environment $E$, (c) the package dependencies ($\text{PDep}$) to produce an updated environment $E'$, (d) the package incorporations to produce an updated environment $E''$, and (e) entities being extended to obtain the environment $E_f$. The overall rule yields a triple formed by the package’s identifier ($Id_p$) and kind ($PK$) and the environment that is produced by processing the PD ($E_f$).

### Table 3.15 Typing Judgements for VCL Package Diagrams

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DE : \text{Id}_p :: (PK, E)$</td>
<td>Well-formed PD yields triple $(Id_p, PK, E)$, comprising the package’s identifier and kind, and an environment</td>
</tr>
<tr>
<td>$DE; E \rightarrow \text{PRef} : E'$</td>
<td>Well-formed sequence of package uses $\text{PRef}$ yields environment $E'$</td>
</tr>
<tr>
<td>$E; \text{Id}_p : \text{PDep} \rightarrow \text{E}'$</td>
<td>Well-formed sequence of package dependencies $\text{PDep}$ yields $E'$</td>
</tr>
<tr>
<td>$E; \text{Id}_p : \text{PRef}_1, \text{Id}_d, \text{Id}_a : \text{E}'$</td>
<td>Identifiers declaring package merge $Id_{d1}$, $Id_{d2}$ and $Id_a$ yield $E'$</td>
</tr>
<tr>
<td>$E; \text{Id}_p : \text{PRef}_2, \text{Id}_d, \text{Id}_a : \text{E}'$</td>
<td>Identifiers declaring package overrides $Id_{d1}$, $Id_{d2}$ and $Id_a$ yield $E'$</td>
</tr>
<tr>
<td>$E; \text{Id}_p : \text{PRef}_1, \text{PRef}_2, \text{Id}_d, \text{Id}_a : \text{E}'$</td>
<td>Well-formed package extends declaration $\text{PExts}$ yields $E'$</td>
</tr>
<tr>
<td>$E; \text{Id}_p : \text{PK} \rightarrow \text{PK}'$</td>
<td>Sequence of incorporated packages $\text{PK}_e$ yields $E'$</td>
</tr>
</tbody>
</table>

### Table 3.16 Type rules for package diagrams (production PD)

$$
\begin{align*}
\text{(Ok PD)} & \quad \text{NoClashes}(\text{PRef}_1, \text{PRef}_2) \quad \text{DE}; E \rightarrow \text{PRef} : E' \\
& \quad E; \text{Id}_p, \text{PRef}_2; \text{PRef}_1 \rightarrow \text{E'} \\
& \quad E; \text{PK} \rightarrow \text{PK}' \\
& \quad \text{DE}; \text{Id}_p : \text{PK} \rightarrow \text{PK}' \\
\end{align*}
$$

### Table 3.17 Type rules for package uses

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\text{PUses } *)$</td>
<td>$DE; E \rightarrow \text{PRef} : E''$</td>
</tr>
<tr>
<td>$DE; E \rightarrow \text{PRef} : E'$</td>
<td>$DE; E \rightarrow \text{PRef} : E'$</td>
</tr>
<tr>
<td>$(\text{PUses NoAlias})$</td>
<td>$\text{Id}_p \in \text{dom } DE$</td>
</tr>
<tr>
<td>$\text{PKG} = DE \text{ Id}_p$</td>
<td>$\rightarrow \text{Id}_p \in \text{dom } E$</td>
</tr>
<tr>
<td>$DE \rightarrow \text{PKG } \text{Id}_p : (Id_p, PK, E')$</td>
<td></td>
</tr>
<tr>
<td>$(\text{PUses Alias})$</td>
<td>$\text{Id}_p \in \text{dom } DE$</td>
</tr>
<tr>
<td>$\text{PKG} = DE \text{ Id}_p$</td>
<td>$\rightarrow (({\text{Id}_p, \text{Id}_a} \subseteq \text{dom } E)$</td>
</tr>
<tr>
<td>$DE \rightarrow \text{PKG } \text{Id}_p : (PK, E')$</td>
<td></td>
</tr>
<tr>
<td>$(\text{PUses *})$</td>
<td>$DE; E \rightarrow \text{PRef} : E'$</td>
</tr>
<tr>
<td>$DE; E \rightarrow \text{PRef} : E'$</td>
<td>$DE; E \rightarrow \text{PRef} : E'$</td>
</tr>
<tr>
<td>$\text{DE}; E \rightarrow \text{PKG } \text{Id}_p : (PK, E')$</td>
<td></td>
</tr>
</tbody>
</table>

31
The rules for processing package uses (Table 3.17) take a sequence of package references, processing each reference in turn, to yield a new environment. The new environment holds bindings for all package types corresponding to the package references. There are two kinds of references: with and without alias. Both rules obtain the VCL package from the diagram environment (DE) and then check it to obtain the package’s environment. The rule then yields an updated environment comprising mappings to the package types and detailed package information for the referenced packages.

### Table 3.18 Type rules for package dependencies (PDep)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PDep *)</td>
<td>E; Id_p; PRef</td>
</tr>
<tr>
<td>(PDep Merge)</td>
<td>E; PRef</td>
</tr>
<tr>
<td>(PDep Override)</td>
<td>E; PRef</td>
</tr>
</tbody>
</table>

Table 3.18 presents the type rules for merge and overrides package dependencies. The rules take a sequence of dependencies, processing each in turn, to yield an updated environment. The appropriate dependency rule, merge or override, is then used depending on the kind of dependency.

### Table 3.19 Type rules for merge dependencies

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PMerge *)</td>
<td>E; Id_p</td>
</tr>
</tbody>
</table>

The type rules for the package merge (Table 3.19) take a source package (Id_p), a target package (Id_p2) and a sequence of set identifiers to produce an updated environment. The rules consider two cases, depending on whether the sequence is empty or not. The non-empty rule (PMerge *) retrieves set definitions for Id_p from environments of source and target package (E_s and E_t), which are retrieved from E. The actual merge is performed by the function mergeVEs (appendix A, section A.3.3). The current environment is then updated with the information of the newly formed set and passed to the rule that processes the remaining sequence of merges.

Like the merge rules, the rules for overrides (Table 3.20) take a source package (Id_p), a target package (Id_p2) and a sequence of set identifiers to produce an updated environment. The non-empty rule (POverrides *) retrieves the types and set information associated with Id_p from environments of source and target package (E_s and E_t), which are retrieved from E. The information associated with Id_p in
Table 3.20 Type rules for overrides dependencies

\[(POverrides \epsilon)\]

\[\begin{align*}
E; Id_p \parallel^{Po}(Id_{ps}, Id_{pt}, \epsilon) & : E \\
\end{align*}\]

(POverrides *)

\[\begin{align*}
E.PE(Id_{ps}) &= (PK_s, E_s) \\
E.PE(Id_{pt}) &= (PK_t, E_t) \\
E_{t}\vdash Id_s : T_s & \quad T_s = \text{Pow Set} \; Id_s \\
E_{t}\vdash Id_s : T_s & \quad E_{t}\vdash Id_s : T_s \\
E_{t}.SE(Id_s) &= (SK_1, DK_1, Id_{p1}, VE) \\
E_{t}.SE(Id_s) &= (SK_2, DK_2, Id_{p2}, \{\}) \\
SI &= (SK_1, DK_1, Id_p, VE) \\
E, Id_s : T_s, Id_s & \rightarrow SI \parallel^{Po}(PId_s, PId_s, Id_s) : E' \\
\end{align*}\]

both environment must be overrides compatible: the target set must have an empty set of properties.

Finally, the actual override is reflected in the way the environment \( E \) is updated with the new bindings for \( Id_p \); the updated environment is then passed to the rule that processes the remaining sequence of overrides.

Table 3.21 Type rules for package incorporations

\[(PIncorporates \epsilon)\]

\[\begin{align*}
E, PK_1 \vdash pinc_\epsilon & : E \\
\end{align*}\]

\[\begin{align*}
\text{(PIncorporates *)} & \quad PK_1 = \text{container} \Rightarrow PK_2 = \text{container} \\
E'' &= \text{transferBindingsForPkgInc}(E_p, E) \\
E''; PK_2 \parallel^{Pext} \text{Transfer} & : E' \\
\end{align*}\]

The rules that process package incorporations (table 3.21) take a sequence of package references to produce a new environment. The non-empty rule (PIncorporates *) retrieves the package information for package identifier \( Id_p \) from the environment \( E \), checks that the package are compatible (if the current package is container, then the incorporated package must also be a container), and transfers all bindings (except those that refer to package types) incorporated package’s environment \( (E_p) \) to the current environment \( E \) to produce an updated environment \( E'' \) using function transferBindingsForPkgInc, (appendix A, section A.3.6). Environment \( E'' \) is then passed to the rule that processes the remaining sequence of incorporated packages.

Table 3.22 Type rules for package extends declaration

\[(PExtends \epsilon)\]

\[\begin{align*}
E; Id_p \parallel^{Pext}(Id_{ps}, Id_{pt}, \epsilon) & : E \\
\end{align*}\]

\[\begin{align*}
\text{(PExtends *)} & \quad E.SE(Id_s) = (SK, SK, Id_o, VE) \\
E \oplus \{Id \mapsto (SK, SK, Id_p, VE)\}^{Pext} \text{extends}(Id_s) & : E' \\
E; Id_p \parallel^{Pext} \text{extends}(Id_s, Id_s) & : E' \\
\end{align*}\]

The rules for processing extensions (table 3.22) take a sequence of extensions, processing each in turn, to yield an updated environment. The empty-sequence rule (PExtends \( \epsilon \)) is straightforward, yielding the current environment \( E \). The non-empty sequence rule (PExtends *) retrieves the set information from
the current environment, and then updates the information changing the package of origin to the current package $Id_p$; this gives ownership to the set to the current package meaning that it can be extended.
Table 3.23 Judgements for type system of VCL Structural Diagrams

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E; \overline{AD}; \overline{Idp} \vdash^{sd} SD :: E'$</td>
<td>Well-formed $SD$ yields $E'$</td>
</tr>
<tr>
<td>$E; \overline{AD}; \overline{Idp} \vdash^{sde} SDE :: E'$</td>
<td>Well-formed sequence of SD elements $SDE$ yields environment $E'$</td>
</tr>
<tr>
<td>$E; \overline{AD}; \overline{Ida} \vdash^{as} \overline{A} :: VE$</td>
<td>Sequence of assertions $\overline{A}$ yields variable environment $VE$</td>
</tr>
<tr>
<td>$E; \overline{AD} \vdash^{gcn} GC :: E'$</td>
<td>Well-formed global constant $GC$ yields environment $E'$</td>
</tr>
<tr>
<td>$E; \overline{AD}; \overline{Ida} \vdash^{cn} \overline{C} :: VE$</td>
<td>Sequence of constants $\overline{C}$ yields variable environment $VE$</td>
</tr>
<tr>
<td>$E; \overline{AD}; \overline{Idp}; T \vdash^{pset} PSet :: E'$</td>
<td>Primary Set $PSet$ yields environment $E'$</td>
</tr>
<tr>
<td>$E \vdash^{ped} PED :: VE$</td>
<td>Sequence of edge definitions $PED$ yields variable environment $VE$</td>
</tr>
<tr>
<td>$E; M \vdash^{mtid} TD :: T$</td>
<td>Designator $TD$ with multiplicity $M$ yields type $T$</td>
</tr>
<tr>
<td>$E; \overline{AD}; \overline{Idp}; T \vdash^{hi} HI :: E'$</td>
<td>$HI$ ($HasIn$) yields environment $E'$</td>
</tr>
<tr>
<td>$E; \overline{T} \vdash \overline{U} :: VE$</td>
<td>Sequence of inside objects $\overline{U}$ yields variable environment $VE$</td>
</tr>
<tr>
<td>$E; \overline{Idp}; T \vdash^{psel} PSet :: E'$</td>
<td>Sequence of inside primary sets $PSet$ yields environment $E'$</td>
</tr>
</tbody>
</table>

3.5 Structural Diagrams

Table 3.23 presents the judgements for structural diagrams (SDs). The first judgement says that a SD is well-formed in the environment $E$ with environment $E'$. The remaining judgements assert well-formedness for the different components of a SD; namely, sequences of structural diagram element (judgement labelled $\vdash^{sde}$), sequences of assertions denoting invariants ($\vdash^{as}$), global constants ($\vdash^{gcn}$), sequences of normal constants ($\vdash^{cn}$), primary sets ($\vdash^{pset}$), sequence of property edge definitions ($\vdash^{ped}$), designators with a multiplicity constraint ($\vdash^{mtid}$), has inside declarations of primary sets ($\vdash^{hi}$), sequence of inside objects ($\vdash^{psel}$) and inside primary sets ($\vdash^{psel}$).

Table 3.24 Type rules for structural diagrams and structural diagram elements

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(Ok SD)$</td>
<td>$E; \overline{Idp} \vdash^{sd} SDE :: E' \quad Acyclic \ E'.SE \quad E'; \overline{AD}; \vdash^{as} \overline{A} :: VE$</td>
</tr>
<tr>
<td>$(SDE *)$</td>
<td>$E; \overline{AD}; \overline{Idp} \vdash^{sde} SDE :: E'' \quad E''; \overline{AD}; \vdash^{sde} :: E'$</td>
</tr>
<tr>
<td>$(SDE Gbl Const)$</td>
<td>$E; \overline{AD}; \overline{Idp} \vdash^{sde} SDE :: E' \quad E; \overline{AD}; \vdash^{sde} :: E'$</td>
</tr>
<tr>
<td>$(SDE RSet Inc)$</td>
<td>$E; \overline{AD}; \overline{Idp}; \overline{T} \vdash^{pset} PSet :: E, E' \quad E; \overline{AD}; \overline{Idp}; \overline{T} \vdash^{set} Idp :: E, E'$</td>
</tr>
<tr>
<td>$(SDE Derived)$</td>
<td>$E \vdash^{def} SetDef :: T \quad E \vdash^{def} PED :: VE \quad E \vdash^{def} pkg { PED } :: E, VE$</td>
</tr>
</tbody>
</table>
### Table 3.25 Type rules for constants

<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Type Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Gbl Const)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} C : VE$</td>
</tr>
<tr>
<td></td>
<td>$E \vdash \text{Id}_{c_n} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp; \text{Id}_{c_n}^{\text{expr}} A : VE_n$</td>
</tr>
<tr>
<td>(Gbl CRef GBl)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} C : VE$</td>
</tr>
<tr>
<td></td>
<td>$E \vdash \text{Id}_{c_n} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp; \text{Id}_{c_n}^{\text{expr}} A : VE_n$</td>
</tr>
<tr>
<td>(Gbl CRef Local)</td>
<td>$E \vdash \text{Id}_d : \text{Pow}\text{ Set}\text{ Id}_d$</td>
</tr>
<tr>
<td></td>
<td>$E; SE(\text{Id}<em>d) = (\text{SK}, \text{DK}, \text{Id}</em>{p_n}, \text{VE}_n)$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} C : VE$</td>
</tr>
<tr>
<td></td>
<td>$E \vdash \text{Id}_{c_n} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp; \text{Id}_{c_n}^{\text{expr}} A : VE_n$</td>
</tr>
</tbody>
</table>

### Table 3.26 Type rules for primary sets

<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Type Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Decl Const)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} C : VE$</td>
</tr>
<tr>
<td></td>
<td>$E \vdash \text{Id}_{c_n} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp; \text{Id}_{c_n}^{\text{expr}} A : VE_n$</td>
</tr>
<tr>
<td>(Decl Const Wconstraint)</td>
<td>$E; T \vdash T : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp; \text{Id}_{c_n}^{\text{expr}} A : VE_n$</td>
</tr>
<tr>
<td>(Cnts epsilon)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} C : VE$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp; \text{Id}_{c_n}^{\text{expr}} A : VE_n$</td>
</tr>
<tr>
<td>(Cnts set)</td>
<td>$E; \overline{\overline{\text{AD}}}; Id : \perp^{\text{expr}} C : VE$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; Id \vdash T : {\text{Id}_{c_n} : T}$</td>
</tr>
</tbody>
</table>

### Table 3.27 Type rules for property edge definitions

<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Type Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>(PED e)</td>
<td>$E; M \vdash^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(PED *)</td>
<td>$E; M \vdash^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TDOpt)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TDOne)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TDSome)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TDMany)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TDMany)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TDMany)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TDMany)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td>(TMany)</td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
<tr>
<td></td>
<td>$E; \overline{\overline{\text{AD}}}; \perp^{\text{expr}} \text{TD} : T$</td>
</tr>
</tbody>
</table>
### Table 3.28 Type rules for sequences of invariants

<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Type Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\mathcal{A} \epsilon))</td>
<td>(E; \mathcal{A} \mathcal{D}; \text{id}<em>{\epsilon} \vdash \mathcal{A} : \mathcal{V} \mathcal{E}</em>{\text{ass}})</td>
</tr>
<tr>
<td>((\mathcal{A} * ))</td>
<td>(E; \overline{\mathcal{A} \mathcal{D}}; \text{id}<em>{*} \vdash \mathcal{A} : \mathcal{V} \mathcal{E}</em>{\text{ass}})</td>
</tr>
</tbody>
</table>

### Table 3.29 Type rules for has inside declarations

<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Type Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{HasInside } \epsilon))</td>
<td>(E; \sigma_{\epsilon} \mathcal{U} : \mathcal{V} \mathcal{E}_{\text{ass}})</td>
</tr>
<tr>
<td>((\text{HasInside } *))</td>
<td>(E; \sigma_{*} \mathcal{U} : \mathcal{V} \mathcal{E}_{\text{ass}})</td>
</tr>
<tr>
<td>((\text{HasInObjs } \epsilon))</td>
<td>(E; \text{id}<em>{\mathcal{P}, T} \vdash \text{hasIn} {\mathcal{O} \mathcal{P}</em>{\mathcal{S}et}} : \mathcal{V'}, \mathcal{V} \mathcal{E}_{\text{ass}})</td>
</tr>
<tr>
<td>((\text{HasInObjs } *))</td>
<td>(E; \text{id}<em>{\mathcal{P}, T} \vdash \text{hasIn} {\mathcal{O} \mathcal{P}</em>{\mathcal{S}et}} : \mathcal{V'}, \mathcal{V} \mathcal{E}_{\text{ass}})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule Description</th>
<th>Type Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{HasInPsets } \epsilon))</td>
<td>(E; \text{id}<em>{\mathcal{P}, T} \vdash \text{hasIn} {\mathcal{O} \mathcal{P}</em>{\mathcal{S}et}} : \mathcal{V'}, \mathcal{V} \mathcal{E}_{\text{ass}})</td>
</tr>
<tr>
<td>((\text{HasInPsets } *))</td>
<td>(E; \text{id}<em>{\mathcal{P}, T} \vdash \text{hasIn} {\mathcal{O} \mathcal{P}</em>{\mathcal{S}et}} : \mathcal{V'}, \mathcal{V} \mathcal{E}_{\text{ass}})</td>
</tr>
</tbody>
</table>

37
Table 3.30 Judgements for typing of assertion diagrams

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash^{ad} AD : I : T$</td>
<td>$AD$ yields binding $I : T$ in $E$</td>
</tr>
<tr>
<td>$E \vdash \top : (VE_v ; VE_h)$</td>
<td>Sequence of declarations $\top$ yields binding sets $(VE_v , VE_h)$</td>
</tr>
<tr>
<td>$E \vdash^{df} DF : (VE_v ; VE_h)$</td>
<td>Declarations formula atom $DF$ yields binding sets $(VE_v , VE_h)$</td>
</tr>
<tr>
<td>$E \vdash^{afs} AFS : T$</td>
<td>Sequence of formulas $AFS$ is well-formed in $E$</td>
</tr>
<tr>
<td>$E \vdash^{id} \text{Id} : (VE_v ; VE_h)$</td>
<td>Arrows formula source $AFS$ yields type $T$</td>
</tr>
</tbody>
</table>

Table 3.31 Type rules for assertion diagrams

\[
\begin{array}{c|c}
(AD \text{ GBL}) & E \vdash \top : (VE_v ; VE_h) \\
E \vdash^{ad} AD \text{ Id}_A \text{ decls } \{\top\} \text{ pred } \{\top\} : I_{\text{Id}} : \text{Assertion} [VE_v , VE_h] \\
\end{array}
\]

3.6 Assertion Diagrams

The judgements for ADs are listed in Table 3.30. In the judgements’ contexts, $E$ is an environment; the AD rules assume that all relevant ADs have been checked and its information can be found in the environment. The judgements are as follows. The first judgement $\vdash^{ad}$ asserts the well-formedness of some AD, yielding a binding made up of the AD’s identifier and type. The remaining judgements concern either the declarations or predicate compartment of ADs. The declarations judgements include: judgement $\vdash^{id}$, which says that a sequence of declarations $(\top)$ is well-formed and $\vdash^{df}$, which says that a particular declaration formula $(DF)$ is well-formed. The predicate compartment includes judgements for formulas $(\vdash^{afs})$ and arrows formula source $(\vdash^{idf})$.

The typing rules for ADs (table 3.31) consider two cases, corresponding to global (AD GBL) and local ADs (AD LOCAL). The rules are similar: the typing of declarations is followed by the typing of the predicate. The local rule requires the local blob environment, which it retrieves from the blob’s type. The processing of the declaration yields two variable environments: the visible variables $(VE_v)$ and the hidden variables $(VE_h)$. The visible variables are visible in the assertions predicate and to the outside world; the hidden variables are only visible within the assertion.

The type rules for the declarations (table 3.32) build the visible and hidden variable environments. They are as follows:

- Rules $\top$ and $\bot$ handle a sequence of declaration inductively. Rule $\top$ yields the empty variable environments ({}); for both visible and hidden: there are no declarations to process. Rule $\bot$ retrieves the variable environments from the current declaration $(VE_v , VE_h)$ and from the remaining declarations $(VE_v , VE_h)$; the variables environments to be yielded by the rule are then merged (operator $\otimes$), which requires that identifiers in common in the variable environments being combined must be bound to the same type; furthermore, all variables from the visible list $(VE_v)$ are removed in the hidden list (operator $\setminus$).

- Rules $\text{Obj}$ and $\text{Set}$ consider the cases where there is a declaration of a scalar (object) or set. Both rules retrieves a type from the declaration’s type designator $(\text{TD})$ and then yield a visible binding made of the variable’s identifier and appropriate type. Rule $\text{Obj}$ considers whether there is an optional qualifier ; type to yield is optional if there is a qualifier $(\text{OptT})$ or the type derived from the type designator otherwise $(T)$. Rule $\text{Set}$ also considers whether there is a sequence qualifier; type to yield is sequence of there is a qualifier $(\text{SeqT})$ or a powerset otherwise $(\text{PowT})$.  

38
<table>
<thead>
<tr>
<th>( \mathcal{T} )</th>
<th>( \mathcal{T} + )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E \vdash D : (VE_\epsilon ; VE_h) )</td>
<td>( E, VE_\epsilon, VE_h \vdash \mathcal{T} : (VE_\epsilon ; VE_h) )</td>
</tr>
<tr>
<td>( VE_\epsilon = VE_\epsilon \uplus VE_\sigma )</td>
<td>( VE_\sigma = (VE_h \uplus VE_{h_\sigma}) \uplus VE_{h_\sigma} )</td>
</tr>
<tr>
<td>( E \vdash \epsilon ; ({}) )</td>
<td>( E \vdash D \mathcal{T} : (VE_\epsilon ; VE_h) )</td>
</tr>
<tr>
<td>(VD)</td>
<td>(VD)</td>
</tr>
<tr>
<td>( (Q = \text{opt} \land T_I = \text{Opt}T \lor Q = \epsilon \land T_I = T) )</td>
<td>( (Q = \text{opt} \lor Q = \epsilon \land T_I = T) )</td>
</tr>
<tr>
<td>( VE = { T_\mathcal{d}, ; T_I } )</td>
<td>( VE = { T_\mathcal{d}, ; \text{Pow}T } )</td>
</tr>
<tr>
<td>( \forall HQ = \text{hidden} \land VE_\epsilon = {} \land VE_h = VE )</td>
<td>( \forall HQ = \text{hidden} \land VE_\epsilon = {} \land VE_h = VE )</td>
</tr>
<tr>
<td>( E \vdash HQ \text{ object } Q T_\mathcal{d}, ; TD : : (VE_\epsilon, VE_h) )</td>
<td>( E \vdash HQ \text{ set } T_\mathcal{d}, ; TD : : (VE_\epsilon, VE_h) )</td>
</tr>
<tr>
<td>(VD Seq)</td>
<td>(VD *)</td>
</tr>
<tr>
<td>( E \vdash TD : T )</td>
<td>( E \vdash TD : T )</td>
</tr>
<tr>
<td>( VE = { T_\mathcal{d}, ; \text{Seq} T } )</td>
<td>( VE = { T_\mathcal{d}, ; \text{Seq} T } )</td>
</tr>
<tr>
<td>( \forall HQ = \text{hidden} \land VE_\epsilon = {} \land VE_h = VE )</td>
<td>( \forall HQ = \text{hidden} \land VE_\epsilon = {} \land VE_h = VE )</td>
</tr>
<tr>
<td>( E \vdash HQ \text{ seq } T_\mathcal{d}, ; TD : : (VE_\epsilon, VE_h) )</td>
<td>( E \vdash HQ \text{ seq } T_\mathcal{d}, ; TD : : (VE_\epsilon, VE_h) )</td>
</tr>
<tr>
<td>(D VD)</td>
<td>(DF)</td>
</tr>
<tr>
<td>( E \vdash VD : : (VE_\epsilon, VE_h) )</td>
<td>( E \vdash DF : : (VE_\epsilon, VE_h) )</td>
</tr>
<tr>
<td>(D DF)</td>
<td>(DF)</td>
</tr>
<tr>
<td>( E \vdash VD : : (VE_\epsilon, VE_h) )</td>
<td>( E \vdash DF : : (VE_\epsilon, VE_h) )</td>
</tr>
<tr>
<td>( E \vdash DF : : (VE_\epsilon, VE_h) )</td>
<td></td>
</tr>
</tbody>
</table>

- Rule DF DF considers the case where the declaration comprises a declaration formula. In this case, the type rule for declaration formulas is called.

Table 3.32 presents the type rules for declaration formulas. The rules are as follows:

- Rules DFA Assertion, DFA OCall, DFA ClCall and DFA PkgCall deal with declaration formula atoms (DFA non-terminal, Fig. 2.10). Rule DFA Assertion considers the case where the construction refers to a normal assertion defined in the same scope (either local or global); rule DFA OCall considers the case where there is a local assertion being called on some object; rule DFA ClCall considers the case where a class assertion is called; rule DFA PkgCall considers the case an assertion from a foreign package is called.

- Rules for declaration formula atoms assume that the AD associated with the assertion being checked has already been type-checked: the assertion’s type can be retrieved from the environment. These rules retrieve the appropriate assertion type from the environment to obtain the assertion’s visible and hidden bindings \((VE_\epsilon \text{ and } VE_h)\). From the assertion’s visible bindings \((VE_\epsilon)\), the rule then builds the visible and hidden bindings for the declaration using function \(\text{con}VE_h\), which takes into account the presence of symbol \(\dagger\), and from these constructed bindings the rule makes the required substitutions according to what is defined in the sequence of renamings \((\mathcal{R})\) using function \(\text{applySubs}\). All it varies in the rules is the way the assertion type is obtained; rule DFA Assertion obtains the assertion type directly from the environment; rule DFA OCall obtains the assertion type from the object’s set; rule DFA ClCall obtains the assertion type from the given set identifier and DFA PkgCall from the given package identifier.

- Rule DF Neg obtains the visible and hidden variables of a negated declarations formula from the enclosed declarations formula.

- Rule DF Bin handles a binary declarations formula combined using a binary operator. The rules obtain the visible and hidden bindings from the two declarations formulas being combined and then merges them using the operator \(\text{mergeves}\).

39
### Table 3.33 Type rules for declaration formulas

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash A : \text{Assertion}[V_{E_0}; V_{E_1}]$</td>
<td>(DFA Assertion)</td>
</tr>
<tr>
<td>$(V_{E_0}, V_{E_1}) = \text{consVEs}(V_{E_0}, [{}])$</td>
<td></td>
</tr>
<tr>
<td>$(V_{E_0}, V_{E_1}) = \text{applySubs}(V_{E_0}, V_{E_0}, [{}])$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash [{} : (V_{E_0}; V_{E_1})$</td>
<td></td>
</tr>
<tr>
<td>$(F \text{ Bin})$</td>
<td></td>
</tr>
<tr>
<td>$(F \text{ AF})$</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.34 Type rules for Formulas (F)

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(f \text{ SF Shaded})$</td>
<td>(F SF Shaded)</td>
</tr>
<tr>
<td>$(F \text{ SF Id})$</td>
<td></td>
</tr>
<tr>
<td>$(F \text{ SF Inside})$</td>
<td></td>
</tr>
<tr>
<td>$(F \text{ Not})$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash F$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash F$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \neg [F]$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \neg [F]$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash [F]$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash [F]$</td>
<td></td>
</tr>
<tr>
<td>$(F \text{ N_ary})$</td>
<td></td>
</tr>
<tr>
<td>$(F \text{ AFS})$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_0 : T_0$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_1 : T_1$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_2 : T_2$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_3 : T_3$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_4 : T_4$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_5 : T_5$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_6 : T_6$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_7 : T_7$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_8 : T_8$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}_9 : T_9$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{10} : T</em>{10}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{11} : T</em>{11}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{12} : T</em>{12}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{13} : T</em>{13}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{14} : T</em>{14}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{15} : T</em>{15}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{16} : T</em>{16}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{17} : T</em>{17}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{18} : T</em>{18}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{19} : T</em>{19}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{20} : T</em>{20}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{21} : T</em>{21}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{22} : T</em>{22}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{23} : T</em>{23}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{24} : T</em>{24}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{25} : T</em>{25}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{26} : T</em>{26}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{27} : T</em>{27}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{28} : T</em>{28}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{29} : T</em>{29}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{30} : T</em>{30}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{31} : T</em>{31}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{32} : T</em>{32}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{33} : T</em>{33}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{34} : T</em>{34}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{35} : T</em>{35}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{36} : T</em>{36}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{37} : T</em>{37}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{38} : T</em>{38}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{39} : T</em>{39}$</td>
<td></td>
</tr>
<tr>
<td>$E \vdash \text{Id}<em>{40} : T</em>{40}$</td>
<td></td>
</tr>
</tbody>
</table>

40
Table 3.35 Type rules for Arrows Formula Source (production AFS)

<table>
<thead>
<tr>
<th>(AFS SE)</th>
<th>(AFS SetId)</th>
<th>(AFS SDef)</th>
<th>(AFS SDef)</th>
<th>(AFSUn Card)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash SE : T$</td>
<td>$E \vdash Id_s : T$</td>
<td>$E \vdash sDef : T$</td>
<td>$E \vdash sDef : T$</td>
<td>$E \vdash # AFS : Int$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(AFS Un Dom)</th>
<th>(AFS Un Dom)</th>
<th>(AFS Un Dom)</th>
<th>(AFS Un Dom)</th>
<th>(AFSUn Dom)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E \vdash AFS : Pow Pair (T_1, T_2)$</td>
<td>$E \vdash AFS : Pow Pair (T_1, T_2)$</td>
<td>$E \vdash AFS : Pow Pair (T_1, T_2)$</td>
<td>$E \vdash AFS : Pow Pair (T_1, T_2)$</td>
<td>$E \vdash AFS : Opt T$</td>
</tr>
</tbody>
</table>

| $E \vdash AFS \rightarrow T$ | $E \vdash AFS \rightarrow T$ | $E \vdash AFS \rightarrow T$ | $E \vdash AFS \rightarrow T$ | $E \vdash AFS \rightarrow T$ |

41
### Table 3.36 Judgements for typing of VCL models

<table>
<thead>
<tr>
<th>Judgement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\vdash vcl$ VCL</td>
<td>VCL is well-formed VCL model</td>
</tr>
<tr>
<td>$DE \vdash_{Pkg} (Id_p, PK, E)$</td>
<td>Well-formed $Pkg$ yields triple $(Id_p, PK, E)$</td>
</tr>
</tbody>
</table>

### Table 3.37 Types rules for overall VCL models made of packages

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(OK VCL)$</td>
<td>$Pkg_{\text{Once}} (VCL)$</td>
</tr>
<tr>
<td>$DE = buildDE (VCL)$</td>
<td>$\text{AyclicPkgs}(DE)$</td>
</tr>
<tr>
<td>$VCL = Pkg$</td>
<td>$DE \vdash_{Pkg} (Id_p, PK, E)$</td>
</tr>
<tr>
<td>$\vdash_{VCL} VCL$</td>
<td>$(OK Pkg)$</td>
</tr>
<tr>
<td>$DE \vdash_{\text{pd}} (Id_p, PK, E)$</td>
<td>$E; \text{AD}; Id_p \vdash_{\text{sd}} (Id_p, PK, E')$</td>
</tr>
<tr>
<td>$DE \vdash_{\text{PD}} SD \text{AD} : (Id_p, PK, E')$</td>
<td></td>
</tr>
</tbody>
</table>

### 3.7 VCL Models

We now describe the typing rules for overall VCL models. The judgements are listed in Table 3.36. The first judgement says that a VCL model is well-formed. The second judgement says that a VCL package ($Pkg$) is well-formed in the diagram environment $DE$.

Table 3.37 presents the type rules for VCL models. The rules are as follows:

- **Rule $\text{OK VCL}$** checks well-formedness of a VCL model. A VCL model is well-formed provided:
  - (a) all its packages are defined once (predicate $Pkg_{\text{Once}}$, appendix A, sec. A.2.4),
  - (b) that the graph formed by the package uses and incorporates dependencies is acyclic (predicate $\text{AyclicPkgs}$, appendix A, sec. A.2.5) in the diagram environment that is built from the VCL model (function $\text{buildDE}$, appendix A, sec. A.3.2),
  - (c) and the current package is well-formed in the diagram environment.

- **Rule $\text{OK Pkg}$** specify well-formedness of a VCL Package. This amounts to type-check PD, SD and ADs. The rule for checking PD ($\vdash_{\text{pd}}$) yields an environment ($E$), which is then used to check the SD and all relevant ADs, resulting in an updated environment.
References


Appendix A

Auxiliary Definitions

This appendix presents the auxiliary definitions that are used in the type system definitions.

A.1 Environment Operators

Several operators manipulate environments. $E_1, E_2$ means that two disjoint environments are combined into one. This is defined as set union for each component of the environments being combined:

$$E_1, E_2 = (\{VE_1, VE_2\}, PE_1 \cup PE_2, SE_1 \cup SE_2, SubE_1 \cup SubE_2)$$

where, $E_1 = (VE_1, PE_1, SE_1, SubE_1) \land E_2 = (VE_2, PE_2, SE_2, SubE_2)$

$VE_1, VE_2$ means that two disjoint variable environments are combined into one. This is defined as set union:

$$VE_1, VE_2 = VE_1 \cup VE_2 \iff \text{dom } VE_1 \cap \text{dom } VE_2 = \emptyset$$

Another operation on variable environments is $\triangleright$, which merges two variable environments. This requires that if there are identifiers in common in both variable environments, then they must be bound to the same type. This operator is defined as a partial function:

$$\triangleright: VE \times VE \to VE$$

This is defined inductively by the following equations:

$$\{\} \triangleright VE = VE$$
$$\{id : T\} \triangleright VE_1 \triangleright VE_2 = VE_1 \triangleright (VE_2 \cup \{id : T\}) \iff id \notin \text{dom } VE_2 \lor VE_2(id) = T$$

We define an operator for performing subtractions on variable environments that require that identifiers in common in both variable environments are bound to the same type. This operator is defined as a partial function:

$$\triangleleft: VE \times VE \to VE$$

This is defined by the following equation:

$$VE_1 \triangleleft VE_2 = VE_1 \setminus VE_2 \iff (\forall id \in (\text{dom } VE_1 \cap \text{dom } VE_2) \bullet VE_1(id) = VE_2(id))$$

$E, VE$ means that a variable environment is added to an environment. This is defined as:
\begin{align*}
E, \text{VE} &= \begin{cases} 
(VE \cup VE, PE, SE, SubE) & \text{if } E = (VE, PE, SE, SubE) \land \text{dom } VE \subseteq \text{dom } VE_
\text{undefined} & \text{otherwise}
\end{cases} \\
E, T_1 \ll T_2 & \text{means that a subtyping tuple is added to an environment. This is defined as:} \\
E, T_1 \ll T_2 &= (VE, PE, SE, SubE \cup \{ T_1 \rightarrow T_2 \}) \quad \text{where, } E = (VE, PE, SE, SubE)
\end{align*}

\begin{align*}
E, Id \rightarrow^\text{se} (SK, DK, Id, VE) & \text{means that a set environment binding is added to an environment. This is defined as:} \\
E, Id \rightarrow^\text{se} (SK, DK, Id, VE) &= \begin{cases} 
(VE, PE, SE \cup \{(SK, DK, Id, VE)\}, SubE) & \text{if } E = (VE, PE, SE, SubE) \land Id \notin \text{dom } E.SE \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{align*}

\begin{align*}
E, Id \rightarrow^\text{pe} (PK, E) & \text{means that a package environment binding is added to an environment. This is defined as:} \\
E, Id \rightarrow^\text{pe} (PK, E) &= \begin{cases} 
(VE, PE \cup \{(PK, E)\}, SE, SubE) & \text{if } E = (VE, PE, SE, SubE) \land Id \notin \text{dom } E.SE \\
\text{undefined} & \text{otherwise}
\end{cases}
\end{align*}

\begin{align*}
E \oplus \text{VE}_2 &= (VE_1 \oplus \text{VE}_2, PE, SE, SubE) \quad \text{where, } E = (VE_1, PE, SE, SubE)
\end{align*}

A.2 Predicates

A.2.1 Predicate Acyclic

The following predicate checks that some graph (or relation) is acyclic.

\text{Acyclic} (R) \iff R \in \{ \text{rel : } X \leftrightarrow X \mid \text{rel} + \cap \text{id } X = \emptyset \}

A.2.2 Predicates IsPEP and IsPEM

\begin{align*}
\text{IsPEP}(PE \overline{PE}) & \iff PE = PEP \land (\overline{PE} = \epsilon \lor \text{IsPEP}(\overline{PE})) \\
\text{IsPEM}(PE \overline{PE}) & \iff PE = PEM \land (\overline{PE} = \epsilon \lor \text{IsPEM}(\overline{PE}))
\end{align*}

A.2.3 Predicate NoClashes

The following predicates check that there are no names clashes in a sequence of package references (grammar of PDs, chapter 2, Fig. 2.7).

\begin{align*}
\text{NoClashes} : F(\overline{PRef}) \\
\text{NoClashes} : F(\overline{PRef}, Id) \\
\text{NoClashes} (\overline{PRef}) & \iff \text{NoClashes}_0 (\overline{PRef}, \emptyset) \\
\text{NoClashes}_0 (\epsilon, S) & \iff Id_p \notin S \lor \text{NoClashes}_0 (\overline{PRef}, S \cup \{Id_p\})
\end{align*}
A.2.4 Predicate PkgsOnce

The following predicate checks that in a VCL model each package is defined only once.

\[
\text{PkgsOnce} : \mathcal{P}(\mathcal{T}_{\text{pkg}})
\]

\[
\text{PkgsOnce}_0 : \mathcal{P}(\mathcal{P}_{\text{ref}}, \hat{\text{id}}) \Rightarrow \text{PkgsOnce}_0(\mathcal{T}_{\text{pkg}}, \{\})
\]

\[
\text{PkgsOnce}_0(\text{Pkg} \mathcal{T}_{\text{pkg}}, S) \Rightarrow \mathcal{T}_{\text{pkg}} = \epsilon \vee (\mathcal{T}_{\text{pkg}} = (\text{PD SD AD}) \wedge \text{Id}_p = \text{getIdOfPD}(\text{PD}) \wedge \text{Id}_p \notin S \wedge \text{PkgsOnce}_0(\mathcal{T}_{\text{pkg}}, S \cup \{\text{Id}_p\}))
\]

A.2.5 Predicate AcyclicPkgs

The predicate AcyclicPkgs checks that the graph of package use and incorporate dependencies is acyclic. This uses the function BuildPkgDepG, which builds the package dependency graph from a diagram environment (DE). Predicate and auxiliary function are defined as follows:

\[
\text{BuildPkgDepG} : \text{DE} \rightarrow (\text{Id} \leftrightarrow \text{Id})
\]

\[
\text{BuildPkgDepG}(\{\}) = \{\}
\]

\[
\text{BuildPkgDepG}(\{\text{Id}_p \rightarrow \text{Pkg} \} \cup \text{DE}) = R \leftrightarrow \text{Pkg} = \text{PD SD AD} \wedge I = \text{getIdsOfIncs}(\text{PD}) \wedge U = \text{getIdsOfUses}(\text{PD}) \wedge R = \{\text{Id}_p\} \times I \cup \{\text{Id}_p\} \times U \cup \text{BuildPkgDepG}(\text{DE})
\]

\[
\text{AcyclicPkgs} : \mathcal{P}(\text{DE}) \Rightarrow \text{Acyclic}(\text{BuildPkgDepG}(\text{DE}))
\]

A.3 Auxiliary Functions

A.3.1 Functions to extract information from PDs

\[
\text{getIdOfPD} : \text{PD} \rightarrow \text{Id}
\]

\[
\text{getIdOfPD}(\text{PK package Id [uses PRef; incorporates PRef;] PDep [PExts]}) = \text{Id}
\]

\[
\text{getIdsOfIncs} : \text{PD} \rightarrow \text{Id}
\]

\[
\text{getIdsOfIncs}(\text{PK package Id [uses PRef; incorporates PRef;] PDep [PExts]}) = \text{getIdsOfPRefSeq}(\text{PRef})
\]

\[
\text{getIdsOfUses} : \text{PD} \rightarrow \text{Id}
\]

\[
\text{getIdsOfUses}(\text{PK package Id [uses PRef; incorporates PRef;] PDep [PExts]}) = \text{getIdsOfPRefSeq}(\text{PRef})
\]

\[
\text{getIdsOfPRefSeq} : \text{PRef} \rightarrow \text{Id}
\]

\[
\text{getIdsOfPRefSeq}(\text{Id}_p [\text{as Id}_a]) = \{\text{Id}_a\} \cup \text{getIdsOfPRefSeq}(\text{PRef})
\]

A.3.2 Function buildDE

\[
\text{buildDE} : \mathcal{T}_{\text{pkg}} \rightarrow \text{DE}
\]

\[
\text{buildDE}(\epsilon) = \{\}
\]

\[
\text{buildDE}(\text{Pkg} \mathcal{T}_{\text{pkg}}) = \{\text{Id}_p \rightarrow \text{Pkg}\} \cup \text{buildDE}(\mathcal{T}_{\text{pkg}}) \leftrightarrow \text{Pkg} = \text{PD SD AD} \wedge \text{Id}_p = \text{getIdOfPD}(\text{PD})
\]

A.3.3 Function getGType

The function getGType gets the greatest type between two types ordered by the subtyping relation:
getGType : $E \times \text{Type} \times \text{Type} \rightarrow \text{Type}$

$$getGType (E, T_1, T_2) = \begin{cases} T_1 & \text{if } E \vdash T_2 <: T_1 \\ T_2 & \text{if } E \vdash T_1 <: T_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

A.3.4 Function superTy

The function superTy gets the super type of some type:

superTy : $E \times \text{Type} \rightarrow \text{Type}$

$$superTy (E, T_1) = T_2 \iff (T_1, T_2) \in E.\text{SubE}$$

A.3.5 Functions producing variable environments (VEs)

The function getVE extracts variable environments from set types:

getVE : $T \times E \rightarrow \text{VE}$

$$getVE (T, E) = \begin{cases} \text{VE} & \text{if } T = \text{Set Ids} \land E.\text{SE}(\text{Ids}) = (SK, DK, Id_p, \text{VE}) \\ \{} & \text{otherwise} \end{cases}$$

The function consVEs constructs a pair of variable environments given an optional imports qualifier and a variable environment (VE). This function simply makes the given VE the first component of the pair if there is an imports qualifier and makes it the second component of the pair otherwise:

consVEs : $\text{VE} \times \upcheck \rightarrow \text{VE} \times \text{VE}$

$$consVEs (\text{VE}, \upcheck) = \begin{cases} (\text{VE}, \{\}) & \text{if } \upcheck = \upcheck \\ (\{\}, \text{VE}) & \text{if } \upcheck = \upcheck \end{cases}$$

The function mergeVEs merges two variable environments (VE). It is a partial function because there is a condition associated with the merge: if there are identifiers in common in both environments, then they map to the same type. The function mergeVEs is as follows:

mergeVEs : $\text{VE} \times \text{VE} \rightarrow \text{VE}$

$$mergeVEs (\text{VE}_1, \text{VE}_2) = \text{VE}_1 \cup \text{VE}_2 \iff \forall \text{id} : \text{Id} \mid \text{id} \in (\text{dom} \text{VE}_1 \cap \text{dom} \text{VE}_2) \bullet \text{VE}_1 \text{id} = \text{VE}_2 \text{id}$$

A.3.6 Function transferBindingsForPkgInc

The function transferBindingsForPkgInc transfer the bindings from one environment to the other. All bindings are transferred except those that refer to a package type.

The auxiliary predicate IsNewBindingOf tells whether some type is a new type of some environment for the purpose of package incorporation (excludes package types).

IsNewBindingOf : $\forall (\text{Id} \times T \times E)$

$$\text{IsNewBindingOf} (\text{Id}, T, E) \iff T \neq \text{PkgId}_p \land \text{Id} \in E.\text{VE}$$

The auxiliary function UpdPkgOfSetType updates the originating package of set types.
\textbf{UpdPkgOfSetType} : \textit{Id} \times T \rightarrow T
\begin{align*}
\text{UpdPkgOfSetType} (\text{Id}_{pm}, T) = \begin{cases} 
\Pow \text{ Set} \text{ Id}_{pm}::\text{Id}_s & \text{If } T = \Pow \text{ Set} \text{ Id}_p::\text{Id}_s \\
T & \text{otherwise}
\end{cases}
\end{align*}

\textbf{transferBindingsForPkgInc} : \textit{Id} \times E \times E \rightarrow E
\begin{align*}
\text{transferBindingsForPkgInc} (\text{Id}_{pm}, \{\}, E) &= E \\
\text{transferBindingsForPkgInc} (\text{Id}_{pm}, \{id \mapsto T\} \cup E, E') &= \\
\Rightarrow \text{IsNewBindingOf} (\text{Id}, T, E') \\
\text{transferBindingsForPkgInc} (\text{Id}_{pm}, \{id \mapsto T\} \cup E, E') &= \text{transferBindingsForPkgInc} (\text{Id}_{pm}, E, E') \\
\Rightarrow \text{IsNewBindingOf} (\text{Id}, T, E')
\end{align*}

A.3.7 Function \textit{getDK}

The function \textit{getDK} extracts the definitional kind:

\begin{align*}
\text{getDK} : [\varnothing] \rightarrow DK \\
\text{getDK} (\varnothing) &= \text{def} \\
\text{getDK} (\epsilon) &= \text{notDef}
\end{align*}

A.3.8 Functions to extract information from ADs

The following functions extract the AD identifier, set identifier and declarations from \textit{ADs}:

\begin{align*}
\text{getIdOfAD} : \text{AD} \rightarrow \text{Id} \\
\text{getIdOfAD} (\text{AD} \text{ Id}_A : [: \text{Id}_s] \text{ decls} \{\overline{D}\} \text{ pred} \{\overline{F}\}) &= \text{Id}_A \\
\text{getSIdOfAD} : \text{AD} \rightarrow \text{Id}_s \\
\text{getSIdOfAD} (\text{AD} \text{ Id}_A : [: \text{Id}_s] \text{ decls} \{\overline{D}\} \text{ pred} \{\overline{F}\}) &= \text{Id}_s \\
\text{getDeclsOfAD} : \text{AD} \rightarrow \overline{D} \\
\text{getDeclsOfAD} (\text{AD} \text{ Id}_A : [: \text{Id}_s] \text{ decls} \{\overline{D}\} \text{ pred} \{\overline{F}\}) &= \overline{D}
\end{align*}

The following functions get the set of ADs that are included in some \textit{AD}:
The following functions look for some AD in a sequence of ADs:

$$
\text{getDepsOfAD} : AD \times \overline{AD} \times Id_+ \rightarrow AD
$$

$$
\text{getDepsOfAD} (AD, \overline{AD}, Id_+) = \text{getADsOfDecls (getDeclsOfAD (AD), \overline{AD}, Id_+)}
$$

$$
\text{getADsOfDecls} : \overline{AD} \times \overline{AD} \times Id_+ \rightarrow AD
$$

$$
\text{getADsOfDecls} (e, \overline{AD}, Id_+) = \{\}
$$

$$
\text{getADsOfDecls} (DF, \overline{AD}, Id_+) = \text{getADsOfDF (DF, \overline{AD}, Id_+)}
$$

$$
\text{getADsOfDecl} : Decl \times \overline{AD} \times Id_+ \rightarrow AD
$$

$$
\text{getADsOfDecl} (DV Id ; TD, \overline{AD}, Id_+) = \{\}
$$

$$
\text{getADsOfDecl} (DF, \overline{AD}, Id_+) = \text{getADsOfDF (DF, \overline{AD}, Id_+)}
$$

$$
\text{getMatchingAD} : AD \times Id \rightarrow \overline{AD}
$$

$$
\text{getMatchingAD} (AD, Id_+) = \{\}
$$

$$
\text{getMatchingAD} (AD, Id_+) = \{\} \quad \text{if getSlAdjAD (AD) } \neq \bot \land \text{getIdOfAD (AD) } = Id_+
$$

$$
\text{getMatchingAD} (AD, Id_+) = \{\} \quad \text{otherwise}
$$

$$
\text{findLADsWithName} : \overline{AD} \times Id \rightarrow AD
$$

$$
\text{findLADsWithName (e, Id_+)} = \{\}
$$

$$
\text{findLADsWithName (AD \overline{AD}, Id_+)} = \text{getMatchingAD (AD, Id_+) } \cup \text{findLADsWithName (\overline{AD}, Id_+)}
$$

### A.3.9 Functions for AD lookup

The following functions look for some AD in a sequence of ADs:

$$
\text{findAD} : \overline{AD} \times Id_+ \times Id_+ \rightarrow AD
$$

$$
\text{findAD (\overline{AD}, \bot, Id_+)} = \text{findGblAD (\overline{AD}, Id_+)}
$$

$$
\text{findAD (\overline{AD}, Id_+, Id_+)} = \text{findLAD (\overline{AD}, Id_+, Id_+)}
$$

$$
\text{findGblAD} : \overline{AD} \times Id_+ \rightarrow AD
$$

$$
\text{findGblAD (AD, Id_+)} = AD \iff \text{getIdOfAD (AD) } = Id_+
$$

$$
\text{findGblAD (AD, \overline{AD}, Id_+)} = AD \iff \text{getIdOfAD (AD) } = Id_+
$$

$$
\text{findLAD} : \overline{AD} \times Id_+ \times Id_+ \rightarrow AD
$$

$$
\text{findLAD (AD, Id_+, Id_+)} = AD \iff \text{getIdOfAD (AD) } \neq Id_+ \lor \text{getIdOfAD (AD) } \neq Id_+
$$

The following function gets the identifier that is associated with an AD, given the name of an assertion and an optional identifier of a constant:

$$
\text{getFAId} : Id_+ \times Id_+ \rightarrow Id_+
$$

$$
\text{getFAId (Id_+, Id_+)} = \begin{cases} 
Id_+ & \text{if } \bot = Id_+ \\
\text{idCnConstraint (CnId, AId)} & \text{otherwise}
\end{cases}
$$
A.3.10 Functions for substitutions

The following functions deal with substitutions in variable environments:

\[ \text{applySubs} : VE \times VE \times P\text{Renaming} \to VE \times VE \]
\[ \text{applySubs}(VE_v, VE_h, Rens) = (\text{doSubs}(VE_v, Rens), \text{doSubs}(VE_h, Rens)) \]

\[ \text{substitute} : VE \times \text{Renaming} \to VE \]

\[ \text{doSubs} : VE \times P_1 \text{Renaming} \to VE \]

\[ \text{substitute}(VE, idn/ido) = \begin{cases} 
  VE & \text{if } ido \notin VE \lor \text{idn} \in VE \\
  (VE \setminus \{(ido, VE \ ido)\}) \cup \{(idn, VE \ ido)\} & \text{otherwise}
\end{cases} \]

\[ \text{doSubs}(VE,) = VE \]
\[ \text{doSubs}(VE, \text{Renaming} \cup Rens) = \text{doSubs}(\text{substitute}(VE, \text{Renaming}), Rens) \]

A.3.11 Function \text{getSIdFrScalarOrCollection}

The following function retrieves a set identifier from types involving set types, which may either denote a scalar or a collection:

\[ \text{getSIdFrScalarOrCollection} : \text{Type} \to Id \]
\[ \text{getSIdFrScalarOrCollection}(\text{Set Id}_s) = \text{Id}_s \]
\[ \text{getSIdFrScalarOrCollection}(\text{Pow Set Id}_s) = \text{Id}_s \]
\[ \text{getSIdFrScalarOrCollection}(\text{Seq Set Id}_s) = \text{Id}_s \]
Appendix B

Alloy Metamodels of VCL Diagrams

B.1 Package Diagrams

-- Name: 'VCL_PD'
--
-- Description:
-- + Defines meta-model of VCL package diagrams (PDs).
--
--========================================================================

module VCL_PD

-- Name: 'Name'
--
-- Description:
-- + Introduces set of labels to be attached Packages
--========================================================================

sig Name {}

-- Name: 'PkgKind'
--
-- Description:
-- + Introduces the package kind; either 'ensemble' or 'container'
--========================================================================

abstract sig PkgKind {}

one sig Ensemble, Container extends PkgKind {}
-- Name: 'VCLPackage'
--
-- Description:
-- + Introduces the labelled VCL package
--==================================================================================
sig VCLPackage {
  name : Name,
  alias : lone Name,
  edges : set PkgEdge
}

-- Each package has its own name (names is an injective function)
fact PkgNamesDistinct {
  all n : VCLPackage.name | one name.n
}

-- The package edges should be anti-reflexive
fact antiReflexiveEdges {
  no (edges.target) & iden
}

-- The overrides and merge relation should be anti-symmetric
fact antiSymmetricOverrides {
  no (edges.target) & -(edges.target)
}

--===============================================================================
-- Name: 'PkgEdgeKind'
--
-- Description:
-- + Introduces package edge kind; either 'overrides' or 'merges'
--===============================================================================
abstract sig PkgEdgeKind {}

one sig Overrides, Merges extends PkgEdgeKind {}

--===============================================================================
-- Name: 'PkgEdge'
--
-- Description:
-- + Introduces the package edge
-- + A package edge comprises a set of names
--===============================================================================
sig PkgEdge {
  names : some Name,
  target : VCLPackage,
  kind : PkgEdgeKind
}

-- All Package edges must have a source
fact allPkgEdgesHaveSource {

all pe : PkgEdge | pe in VCLPackage.edges
}
--================================================================---------
-- Name: 'ExtendsList'
--
-- Description:
-- + Introduces a signature that holds the extends list of a current package
-- + It comprises a set of names
--================================================================---------
sig ExtendsList {
  extElems : set Name
}
--================================================================---------
-- Name: 'CurrPackage'
--
-- Description:
-- + Current package, which can have other packages inside,
-- + and can import other packages
--================================================================---------
one sig CurrPackage extends VCLPackage {
  inside : set VCLPackage,
  imports : set VCLPackage,
  pkind : PkgKind,
  extendsLs : lone ExtendsList
}{
  -- The current package mustn't have an alias
  no alias
}
--
-- Current packages cannot have override or merge edges
fact CurrPackageCannotOverrideOrMerge { no (CurrPackage.edges.kind & (Overrides+Merges)) }
--
-- Current packages cannot be target of a dependency.
fact CurrPackageCannotBeOverridenOrMerged { no (edges.target).CurrPackage }
--
-- The 'inside' relation should be irreflexive
fact irreflexiveIncorporates {
  no inside & iden
}
--
-- The 'imports' relation should be irreflexive

54
fact irreflexiveImports {
    no imports & iden
}

--
-- Imported packages should not have any edges
fact noEdgesForImported {
    no imports & edges.target
}

--
-- Packages cannot be imported and incorporated
fact noImportsAndIncorporates {
    no imports & inside
}

--=========================================================================--
-- Name: 'PackageDiagram'
--
-- Description:
-- + Introduces the package diagram.
-- + A single current packages and package edges.
--=========================================================================--
sig PackageDiagram {
    defines : CurrPackage
}

-- All packages must be defined in a package diagram
all p : (VCLPackage-CurrPackage) | p in defines.inside

B.2 Common

--=========================================================================--
-- Name: 'VCL_Common'
--
-- Description:
-- + Common entities of VCL ADs and SDs
--=========================================================================--
module VCL_Common

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

-- Signature of all names
sig Name {}
abstract sig SetElement {
}

sig VCLObject extends SetElement {
    id : Name
}

sig Pair extends SetElement {
    idElem1 : Name,
    idElem2 : Name
}

sig Assertion {
    idAssertion : Name
}

sig TypeDesignator {
    + Defines a designator for types.
}
abstract sig TypeDesignator {
}

-- Name: 'TypeDesignator', 'TypeDesignatorNat'
-- Description:
-- + Defines a type designator naturals and integers.
sig TypeDesignatorInt, TypeDesignatorNat extends TypeDesignator {
}

-- Name: 'TypeDesignatorId'
-- Description:
-- + Defines a designator of blobs with an identifier.
sig TypeDesignatorId extends TypeDesignator {
    id : Name
}

-- Name: 'PropEdge'
-- Description:
-- + Defines property edges with a source and a target.
abstract sig PropEdge {
    op : EdgeOperator,
    target : Expression,
}

-- Name: 'PropEdgePred'
-- Description:
-- + Defines property edges attached to predicate elements.
sig PropEdgePred extends PropEdge {
    unop : lone EdgeOperatorUnary,
    designator : lone Name
}

-- 'op' must be a 'EdgeOperatorPred'
    op in EdgeOperatorBin
}

-- Name: 'PropEdgeMod'
-- Description:
-- + Defines the property edge modifier that applies some operation to
-- the source.
--=========================================================================
sig PropEdgeMod extends PropEdge {
}{
    -- 'op' must be a 'EdgeOperatorMod'
    op in EdgeOperatorMod
}
--=========================================================================

-- Name: 'EdgeOperator'
--
-- Description:
-- + Defines edge operator used in edges.
--=========================================================================
abstract sig EdgeOperator {
}
--=========================================================================

-- Name: 'EdgeOperatorBin'
--
-- Description:
-- + Defines edge operator used in predicate edges.
--=========================================================================
abstract sig EdgeOperatorBin extends EdgeOperator{
}
--=========================================================================

-- Name: 'EdgeOperatorMod'
--
-- Description:
-- + Defines edge operator used in modifier edges.
--=========================================================================
abstract sig EdgeOperatorMod extends EdgeOperator{
}
--=========================================================================

-- Name: 'EdgeOperatorUnary'
--
-- Description:
-- + Defines edge operator used in modifier edges.
--=========================================================================
abstract sig EdgeOperatorUnary extends EdgeOperator{
}
--=========================================================================

-- Name: 'EdgeOperatorEq', 'EdgeOperatorIn', 'EdgeOperatorSubsetEQ'
-- 'EdgeOperatorLT', 'EdgeOperatorLEQ', 'EdgeOperatorGT', 'EdgeOperatorGEQ'
--
-- Description:
-- + Defines different kinds of edge operators.
-- + Eq (=), Neq (≠), In (￿), LT, (<), LEQ (⪯), GT (>), GEQ (⪰)
-- + SubsetEQ (⪯)
one sig EdgeOperatorEq, 
EdgeOperatorNEq, 
EdgeOperatorIn, 
EdgeOperatorLT, 
EdgeOperatorLEQ, 
EdgeOperatorGT, 
EdgeOperatorGEQ, 
EdgeOperatorSubsetEQ 
  extends EdgeOperatorBin {
}

-- Name: 'EdgeOperatorDRES', 'EdgeOperatorRRES'
--
-- Description:
-- + Edge Operators used in property edge modifiers.
-- + DRES (￿, domain restriction), and RRES (￿, range restriction)
-- + DSUB (, domain subtraction) and RSUB (, range subtraction)

one sig EdgeOperatorDRES, 
EdgeOperatorRRES,
  EdgeOperatorDSUB, 
EdgeOperatorRSUB 
extends EdgeOperatorMod {
}

-- Name: 'EdgeOperatorCARD', 'EdgeOperatorTHE'
--
-- Description:
-- + Unary edge operator used in predicate property edges
-- + CARD (#, cardinality)
-- + THE (, the)

one sig EdgeOperatorCARD, EdgeOperatorTHE 
  extends EdgeOperatorUnary {
}

-- Name: 'Num'
--
-- Description:
-- + String representing natural numbers.

sig Num {}

-- Name: 'Expression'
--
-- Description:
-- + Defines expressions associated with property edges.
abstract sig Expression {
}

--=========================================================================
-- Name: 'ObjExpression'
--
-- Description:
-- + Defines an object expression.
--=========================================================================
abstract sig ObjExpression extends Expression {
}

--=========================================================================
-- Name: 'ObjExpressionId'
--
-- Description:
-- + Defines object expressions comprising an identifier (a name).
--=========================================================================
sig ObjExpressionId extends ObjExpression {
    eid : Name
}

--=========================================================================
-- Name: 'ObjExpressionNum'
--
-- Description:
-- + Defines expressions comprising a number.
--=========================================================================
sig ObjExpressionNum extends ObjExpression {
    num : Num
}

--=========================================================================
-- Name: 'ObjExpressionUMinus'
--
-- Description:
-- + Defines unary minus expression (-e).
--=========================================================================
sig ObjExpressionUMinus extends ObjExpression {
    e : ObjExpression
}

--=========================================================================
-- Name: 'ObjExpressionBin'
--
-- Description:
-- + Defines expressions that can be combined with binary operators.
--=========================================================================
abstract sig ObjExpressionBin extends ObjExpression {
    e1, e2 : ObjExpression,
}
op : ObjExpBinOp
}
  e1 != e2
}

-- Name: 'ObjExpressionPar'
--
-- Description:
-- + Defines expressions that can be placed within parenthesis.
--
abstract sig ObjExpressionPar extends ObjExpression {
  e : ObjExpression,
}

-- Name: 'ObjExpBinOp'
--
-- Description:
-- + Infix operators for sum (+), subtraction (-), product (*), div (/).
--
abstract sig ObjExpBinOp {}

one sig ExpBinOpPlus,
  ExpBinOpMinus,
  ExpBinOpTimes,
  ExpBinOpDiv extends ObjExpBinOp {}

-- Name: 'BlobExpression'
--
-- Description:
-- + Defines a blob expression.
--
abstract sig BlobExpression extends Expression {
}

-- Name: 'BlobExpressionID'
--
-- Description:
-- + Defines a blob expression defined using a blob designator.
--
sig BlobExpressionID extends BlobExpression {
  bd : TypeDesignator
}

-- Name: 'BlobExpressionEmpty'
--
-- Description:
-- + Defines a blob that is shaded to represent the empty set.
sig BlobExpressionEmpty extends BlobExpression {
}

--=========================================================================
-- Name: 'BlobExpressionCard'
--
-- Description:
-- + Defines a blob with a cardinality unary operator attached.
--=========================================================================
sig BlobExpressionCard extends BlobExpression {
  blExp : BlobExpression
}

--=========================================================================
-- Name: 'BlobDef'
--
-- Description:
-- + Defines a blob definition (symbol ).
--=========================================================================
sig BlobDef {
  bdop : BlobDefOp, -- optional blob def operator
  insideExp : BlobInsideExpression
}

--=========================================================================
-- Name: 'BlobDefOp'
--
-- Description:
-- + Defines blob def operators
-- + Domain operator is represented as symbol
-- + Range operator is represented as symbol
-- + None represents no symbol
-- + Union operator is represented as symbol
-- + Intersection operator is represented as symbol
-- + Cross product operator is represented as symbol
-- + Set difference operator is represented as symbol
--=========================================================================
abstract sig BlobDefOp {
}

one sig BlobDefOpDomain,
  BlobDefOpRange,
  BlobDefOpNone,
  BlobDefOpUnion,
  BlobDefOpIntersection,
  BlobDefOpCrossProduct,
  BlobDefOpSetMinus
extends BlobDefOp {
}

--=========================================================================
-- Name: 'BlobExpressionDef'
--
-- Description:
-- + Defines a blob expression defined using a blob definition.
--=========================================================================

sig BlobExpressionDef extends BlobExpression {
  def : BlobDef
}

-- Name: 'BlobInsideExpression'
--
-- Description:
-- + Expression inside the blob def
--=========================================================================
abstract sig BlobInsideExpression {
}

-- Name: 'InsideExpBlDs'
--
-- Description:
-- + Expression inside the blob def
--=========================================================================
sig InsideExpBlDs extends BlobInsideExpression {
  blobDefs : seq BlobDef
}

-- Name: 'InsideDef'
--
-- Description:
-- + Definition of the blob def
-- + Either a constrained blob or a set extension
--=========================================================================
abstract sig InsideDef extends BlobInsideExpression {
}

-- Name: 'ConstrainedBlob'
--
-- Description:
-- + Defines a blob with restrictions (constraints).
--=========================================================================
sig ConstrainedBlob extends InsideDef {
  bd : TypeDesignator,
  pes : seq PropEdge -- 0 or more predicate property edges
}

fact PropEdgesOfConstrainedBlobAreOfSomeKind {
all be : ConstrainedBlob |
  all disj pe1, pe2 : univ.(be.pes) |
  pe1+pe2 in PropEdgePred || pe1+pe2 in PropEdgeMod }

-- Name: 'BlobExtension'
--
-- Description:
--  + Defines a blob extensionally by listing its members.
--

sig BlobExtension extends BlobInsideExpression {
  elems : some SetElement
}

B.3 Structural Diagrams

-- Name: 'VCL_SD'
--
-- Version: 3.7
--
-- Description:
--  + Defines meta-model of VCL structural diagrams (SDs).
--

module VCL_SD

open VCL_Common as c

-- Name: 'Bool'
--
-- Description:
--  + Signature of booleans: 'True' or 'False'.

abstract sig Bool {}

one sig True, False extends Bool {}

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

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

one sig MStar {}

sig MRange extends Mult {
  -- lower and upper bounds for 'range' multiplicities.
  lb : Int,
  ub : (Int+MStar)
}{
  -- lower and upper bounds must be greater or equal than 0
  -- and 'ub' greater or equal than 'lb'.
  lb >= 0 && (ub = MStar || ub >= lb)
}

-- Name: 'SDElem'
--
-- Description:
--  + Introduces the labelled structural diagram element.
--  + To be extended by 'Blob', 'Object', 'Edge'.
abstract sig SDElem {
  name : Name -- a modelling element has a name (a label).
}

-- Name: 'ConstantKind'
--
-- Description:
--  + Indicates kind of constant: reference or definition
abstract sig ConstantKind {
}

one sig ConstReference, ConstDefinition extends ConstantKind {
}

-- Name: 'Constant'
--
-- Description:
--  + Represents constants. A constant has a type (field 'type').
--  + Constants can be 'local' or 'global'.
--  + A constant definition has a type
--  + A constant may have a defining assertion, which defines its value
--  + A constant reference has an 'origin' 'Blob or package'

sig Constant extends SDElem {
kind : ConstantKind,  
type : lone Name,  
definition : lone Assertion,  
origin : lone Name
}

-- A constant either has a type or an origin
kind in ConstDefinition => one type && no origin
kind in ConstReference => one origin && no type

--============================================================================
-- Name: 'RelEdge' (Relational Edge)
--
-- Description:
-- + Blob relational edges are binary edges connecting blobs.
-- + They have multiplicities at each end of edge.
--============================================================================
sig RelEdge extends SDElem {
   source, target : Blob,  
   sourceMult, targetMult : Mult,
}

   -- Relation edges cannot have multiplicities of type sequence
   not (sourceMult+targetMult) in MSeq

--============================================================================
-- Name: 'InNode'
--
-- Description:
-- + Nodes that can be inside primitive blobs.
-- + A Node can either hold a 'VCLObject' or a 'PrimaryBlob'.
--============================================================================
sig InNode {
   node : (VCLObject+PrimaryBlob)
}

   -- Each 'InNode' has its own referenced node (object or blob)
fact NodesNotShared {
   all n : (VCLObject+PrimaryBlob) | (some node.n)  
   => one node.n
}

--============================================================================
-- Name: 'Blob' (Blob Definitions)
--
-- Description:
-- + Defines a global blob definition.
-- + It's characterised by inside property.
--
abstract sig Blob extends SDElem {
}

--
-- This is a well-formedness constraint to rule out redundant definitions!
-- 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)
--}

-- Name: 'IntBlob' (Integer Blob)
--
-- Description:
-- + Defines a blob representing the integers

one sig IntBlob extends Blob {}

-- Name: 'NatBlob' (Natural numbers Blob)
--
-- Description:
-- + Defines a blob representing the natural numbers

one sig NatBlob extends Blob {}

abstract sig BlobKind {}

-- Name: 'Value', 'Domain'
--
-- Description:
-- + Defines two blob kinds: 'value' and 'domain.

one sig Value, Domain extends BlobKind {}

-- Name: 'BlobWithProps'
--
-- Description:
-- + Defines a blob with local properties

abstract sig BlobWithProps extends Blob {
   lProps : set PropEdgeDef
}
-- Name: 'PrimaryBlob'
--
-- Description:
-- + Defines a primary blob
-- + A Primary blob can have other primary blobs inside.
--
sig PrimaryBlob extends BlobWithProps {
  hasInsideB : set PrimaryBlob
}

-- 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 ^\{hasInsideB\} & iden
}

-- An object should be in at most one blob (the inverse of the relation is a partial function)
fact blobInAtMostOneBlob {
  all b : PrimaryBlob | lone b.~hasInsideB
}

-- Name: 'PrimaryBlobDef'
--
-- Description:
-- + Defines a primary blob definition
-- + A Primary blob can have blobs ad objects inside.
--
sig PrimaryBlobDef extends PrimaryBlob {
  kind : BlobKind,
  isDefBlob : Bool, -- (symbol )
  hasInsideO : set VCLObject,
  lInvariants : set Assertion,
  lConstants : set Constant,
}

fact objInAtMostOneBlob {
  all n : VCLObject | lone n.^hasInsideO
}

-- Each 'Blob' has its own set of local invariants.

68
-- Or local invariants are not shared.
fact LInvariantsNotShared {
  all c : Assertion | (some lInvariants.c)
    => one lInvariants.c
}

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

-- Definitional blobs must have things inside.
fact DefBlobsHasThingsInside {
  all b : isDefBlob.True | #b.hasInsideO > 0 || #b.hasInsideB > 0
}

-- Each domain blob can contain other domain blobs only
-- and they can be inside of domain blobs only.
fact DBlobHasDBlobsInside {
  all b : PrimaryBlob | b.kind = Domain
    => (b.hasInsideB) in kind.Domain & hasInsideB.b in kind.Domain
}

-- Name: 'BlobReference'
-- Description:
-- + Defines a blob representing a reference to a blob defined in other
-- package
-- + Reference blobs have their names preceeded by the symbol '↑'
-- + Blob references have a reference to a package. The package
-- name is sepearated from the blob's name using '::'

-- Name: 'PkgBlob'
-- Description:
-- + Defines a package blob
-- + Package blobs are represented in bold and double-lined.

69
sig PropEdgeDef extends SDElem {
  peTarget : Blob,
  mult : Mult
}

-- a PropEdgeDef cannot have its blob or his inside blobs as target
-- not (peTarget in ((this.~lProps) + (this.~lProps).^(hasInsideB)))

-- Each 'Blob' has its own set of property edge definitions
-- Or property edges are not shared. All property edges belong to some blob
fact propEdgesNotSharedAndBelongToSomeBlob {
  all pe : PropEdgeDef | one lProps.pe
}

fun nameOf (elem : SDElem + Assertion) : Name {
  elem in SDElem implies elem.name else elem.idAssertion
}

-- Local Names in the scope of a 'Blob' must be unique
--
fact LocalNamesAreUnique {
  all b : Blob |
  all e1, e2 : (b.1Constants+b.1Invariants+b.1Props+(b.hasInsideO)) |
  nameOf [e1] = nameOf [e2] => e1 = e2
}

-- All global names must be unique

fun nameOf (b : Blob + Assertion) : Name {
  (b.1Constants+b.1Invariants+b.1Props+(b.hasInsideO)) |
  nameOf [b] = nameOf [b] implies b = b
}
-- Name: 'DerivedBlob'
--
-- Description:
-- + Defines a derived blob
-- + Derived blobs make use of symbol '  

--=========================================================================
sig DerivedBlob extends Blob {
    definition : BlobDef
}
--=========================================================================

-- Name: 'SDiag'
--
-- Description:
-- + Defines a derived blob
-- + Derived blobs make use of symbol '  

--=========================================================================
sig SDDiag {
    sdelems : set SDElem,
    invs : set set Assertion
}

B.4 Common Assertion and Contract Diagrams
--=========================================================================

-- Name: 'VCL_AD_CD_Common'
--
-- Version: 1.5
--
-- Description:
-- + This module provides definitions common to both ADs and CDs.
--=========================================================================

open VCL_Common as c
--=========================================================================

-- Name: 'Bool'
--
-- Description:
-- + Signature of booleans: 'True' or 'False'.
--=========================================================================

abstract sig Bool {}  
one sig True, False extends Bool {}  

-- Name: 'Decl'
--
-- Description:
-- + Defines a declaration of an assertion diagram.
abstract sig Decl {
}

--===============================================================================
-- Name: 'TypedDecl'
--
-- Description:
-- + Defines a typed declaration of an assertion diagram.
-- + A typed declaration has name, which represents a variable's id
-- + Typed declarations define variables, either objects or blobs
--===============================================================================
abstract sig TypedDecl extends Decl {
    dName : Name, -- Name of declaration
    dTy : TypeDesignator // Type of declaration
}

--=========================================================================
-- Name: 'DeclObj'
--
-- Description:
-- + Defines declarations of objects.
-- + Declarations of objects are represented as objects (rectangles).
-- + field optional indicates whether declaration is optional or not
-- + If optional is true, then '?' precedes the object's type.
--=========================================================================
sig DeclObj extends TypedDecl{
    optional : Bool
}

--=========================================================================
-- Name: 'DeclBlob'
--
-- Description:
-- + Defines declarations of blobs.
-- + Blobs are represented as blobs (rectangles with round corners)
-- + If sequence boolean is true then '[]' precedes the object's type.
--=========================================================================
sig DeclBlob extends TypedDecl {
    isSequence : Bool
}

--=========================================================================
-- Name: 'DeclFormula'
--
-- Description:
-- + Defines a declaration reference formula.
-- + This enables declaration references (either assertions or contracts)
-- to be combined using logical operators.
--=========================================================================
abstract sig DeclFormula extends Decl {

abstract sig DeclRefKind {}
abstract sig DeclFormulaAtom extends DeclFormula {
    refId : Name, -- Id of referenced assertion or contract
    refKind : DeclRefKind, -- Kind of reference (simple, call, new call, package)
    import : Bool, -- Whether import symbol is present or not
    origin : lone Name, -- optional origin (calling object, blob or package)
    renameExp : set RenamingExp -- a set of renaming expressions
}

--=========================================================================
-- Name: ' DeclFormulaNot'
--
-- Description:
-- + Defines a declaration negation formula
--=========================================================================

sig DeclFormulaNot extends DeclFormula {
    df : DeclFormula
}

--=========================================================================
-- Name: 'DeclFormulaBinOp'
--
-- Description:
-- + Defines a binary Formula operator for declaration references.
--=========================================================================

abstract sig DeclFormulaBinOp {
}

--=========================================================================
-- Name: DFImplies, DFAnd, DFOr, DFEquiv, DFComp
--
-- Description:
-- + Defines formulas for implication (f1[*]f2), conjunction ([f1*f2]),
-- disjunction ([f1|f2]), equivalence ([f1=f2])
-- and sequential composition ([f1;f2]).
--=========================================================================

one sig DFImplies, DFAnd, DFOr, DFEquiv, DFComp extends DeclFormulaBinOp {
}

--=========================================================================
-- Name: ' DeclFormulaBin'
--
-- Description:
-- + Defines a declaration binary formula
-- + This supports the logical operators , ,
--=========================================================================

sig DeclFormulaBin extends DeclFormula {
    dFrml1, dFrml2 : DeclFormula,
    dfop : DeclFormulaBinOp
}

--=========================================================================
-- Name: 'FormulaSource'

74
abstract sig FormulaSource { }

-- Name: 'FormulaSourceElement'
-- Description:
-- + Defines source formula of type object
-- + 'elem' indicates the 'SetElement' either object or pair

sig FormulaSourceElem extends FormulaSource {
  elem : SetElement
}

-- Name: 'FormulaSourceBlob'
-- Description:
-- + Defines source formula of type blob

abstract sig FormulaSourceBlob extends FormulaSource { }

-- Name: 'FormulaSourceBlobId'
-- Description:
-- + Defines source formula of type blob identifier
-- + 'bId' indicates identifier of the blob

sig FormulaSourceBlobId extends FormulaSourceBlob {
  bId : Name
}

-- Name: 'FormulaSourceBlobDef'
-- Description:
-- + Defines source formula of type blob definition
-- + 'blDef' holds blob definition

sig FormulaSourceBlobDef extends FormulaSourceBlob {
  blDef : BlobDef
}

-- Name: 'FormulaSourceUOp'
-- Description:
--  + Defines a unary Formula operator for a formula source.
--=========================================================================
abstract sig FormulaSourceUOp {
}
--=========================================================================
-- Name: FSBCardinality, FSBDom, FSBRan
--
-- Description:
--  + Symbol of Formula source operator cardinality is #
--  + Symbol of Formula source operator domain is '￿'
--  + Symbol of Formula source operator range is '￿'
--  + Symbol of Formula source operator the is '￿'
--========================================================================
one sig FSBCardinality, FSBDom, FSBRan , FSBThe extends FormulaSourceUOp {
}
--=========================================================================
-- Name: 'FormulaSourceUnary'
--+--
-- Description:
--  + Defines source formula with unary operator
--  + Let 'O' be a blob, this construction is expressed as # [O]
--=========================================================================
sig FormulaSourceUnary extends FormulaSource {
    operator : FormulaSourceUOp,
    frmlSrc : FormulaSource
}
--=========================================================================
-- Name: 'DeclCompartment'
--
-- Description:
--  + Defines a declarations compartment
--  + Comprises a set of declarations and a set of decls formula
--=========================================================================
sig DeclCompartment {
    decls : set Decl,
    declFrmls : set DeclFormula
}

B.5 Assertion Diagrams

-- Name: 'VCL_AD'
--
-- Description:
--  + Module defining the meta-model of VCL assertion diagrams.
open VCL_Common as c
open VCL_AD_CD_Common as d

-- Name: 'AD'
-- Description:
--   + Defines what an assertion diagram is.
abstract sig AD {
    aName : Name,
}

-- Name: 'VAD'
-- Description:
--   + Defines what a visual assertion diagram is.
--   + A VAD supports the visual expression of ADs.
sig VAD extends AD {
    declarations : DeclCompartment,
    predicate : set Formula
}

-- Name: 'TAD'
-- Description:
--   + Defines what a textual assertion diagram is.
--   + A TAD supports the visual expression of an assertion using the target
--     language (e.g Z).
sig TAD extends AD {
    txtAssertion : String
}

-- Name: 'Formula'
-- Description:
--   + Defines a Formula.
abstract sig Formula {
}

--
-- Name: 'FormulaNot'
--
-- Description:
-- + Defines a not Formula (¬[f]).
--=========================================================================
sig FormulaNot extends Formula {
   f: Formula
}
--=========================================================================

-- Name: 'FormulaBin'
--
-- Description:
-- + Defines a binary Formula.
--=========================================================================
sig FormulaBin extends Formula {
   f1, f2: Formula,
   bop : FormulaBinOp
}
{f1 != f2}
--=========================================================================  

-- Name: 'FormulaBinOp'
--
-- Description:
-- + Defines a binary Formula operator.
--=========================================================================
abstract sig FormulaBinOp {
}
--=========================================================================  

-- Name: FImplies, FAnd, FOr, FEquiv
--
-- Description:
-- + Defines formulas for implication ([f1] [f2]), conjunction ([f1] [f2]),
-- disjunction ([f1] [f2]) and equivalence ([f1] [f2]).
--=========================================================================  
one sig FImplies, FAnd, FOr, FEquiv extends FormulaBinOp {
}
--=========================================================================  

-- Name: 'ArrowsFormula'
--
-- Description:
-- + Defines an arrows formula
-- + Made of predicate property edges
-- + With a source, which can either be: obj, blob or pair
--=========================================================================
sig ArrowsFormula extends Formula {
   source : FormulaSource,
   pes : some PropEdgePred
}
abstract sig BlobFormula extends Formula {
}

-- Name: 'BlobFormulaDef'
--
-- Description:
-- + Defines a 'Blob' formula using a blob definition (symbol )

sig BlobFormulaDef extends BlobFormula {
    shaded : Bool, -- blob may be shaded to mean empty set
    bid : lone TypeDesignator,-- the optional blob designator
    bdef : BlobDef -- the blob definition
}

-- Name: 'BlobFormulaSubset'
--
-- Description:
-- + Defines a 'Blob' formula defined using a subset definition.

sig BlobFormulaSubset extends BlobFormula {
    bid : TypeDesignator,
    hasInside : BlobExpression
}

-- Name: 'BlobFormulaShaded'
--
-- Description:
-- + Defines a 'Blob' formula defined using shading.

sig BlobFormulaShaded extends BlobFormula {
    bid : TypeDesignator
}