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

Chapter 1

Introduction

This document present a type system for the Visual Contract Language (VCL) [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*[1] [AGK11]. The following gives some background on VCL and an outline of the overall document.

1.1 Background: The Visual Contract Language (VCL)

VCL [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 [APS05, Amá07], a semantic domain of object orientation for the language Z [Spi92, ISO02].

1.1.1 VCL Diagrams

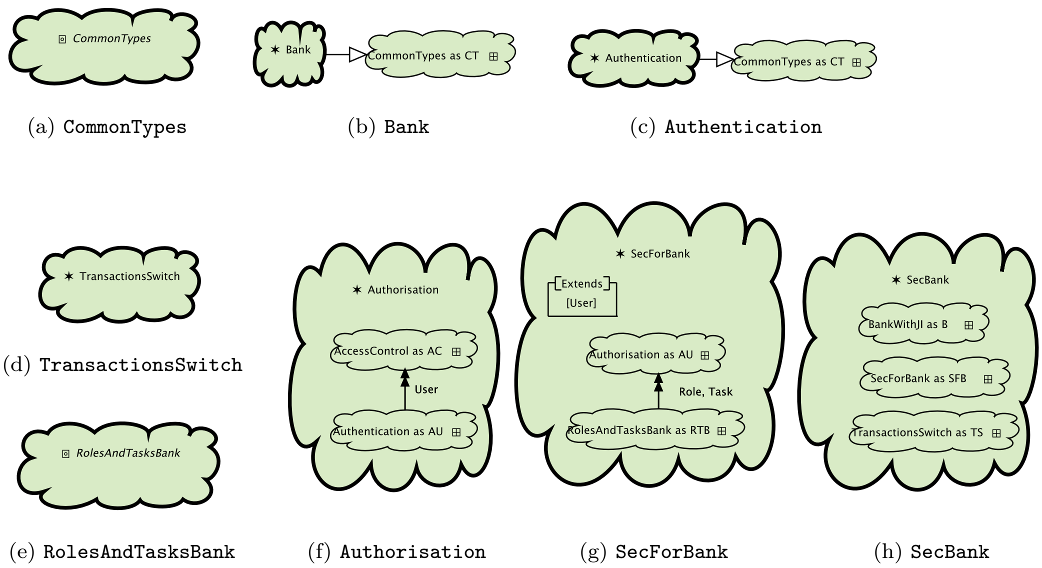
VCL's diagram suite comprises *package*, *structural*, *behaviour*, *assertion* and *contract* diagrams.

Package Diagrams

Package diagrams (PDs) define VCL packages, coarse-grained modules representing concerns. Package are represented as *clouds* because they define a world of their own. Sample PDs from the secure simple bank case study [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 ⧇) or *ensemble* (symbol ✶). 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.

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



(a) `CommonTypes` (b) `Bank` (c) `Authentication` (d) `TransactionsSwitch` (e) `RolesAndTasksBank` (f) `Authorisation` (g) `SecForBank` (h) `SecBank`

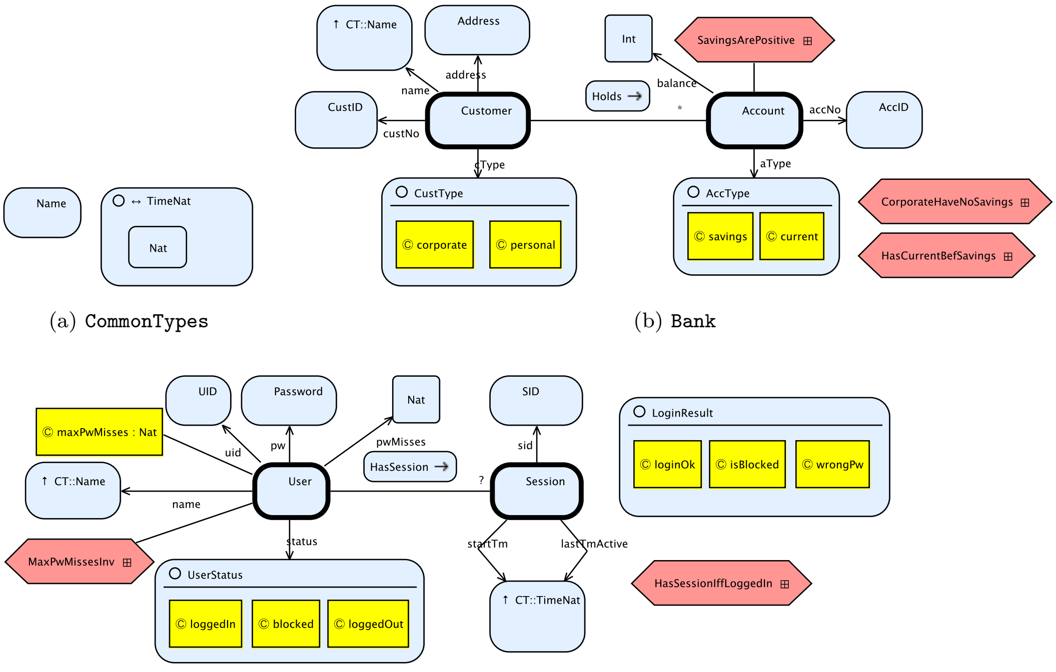
Figure 1.1: Sample package diagrams of *secure simple bank* (from [Amá11])

- A current package may defined 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 ◯ are *definitional*. This means that they are defined by what they enclose. Sets that include symbol ◯ followed by ↔ are *derived*. This means that they are



(a) `CommonTypes`

(b) `Bank`

(c) `Authentication`

Figure 1.2: Sample structural diagrams of *secure simple bank* (from [Amá11])

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`[2].

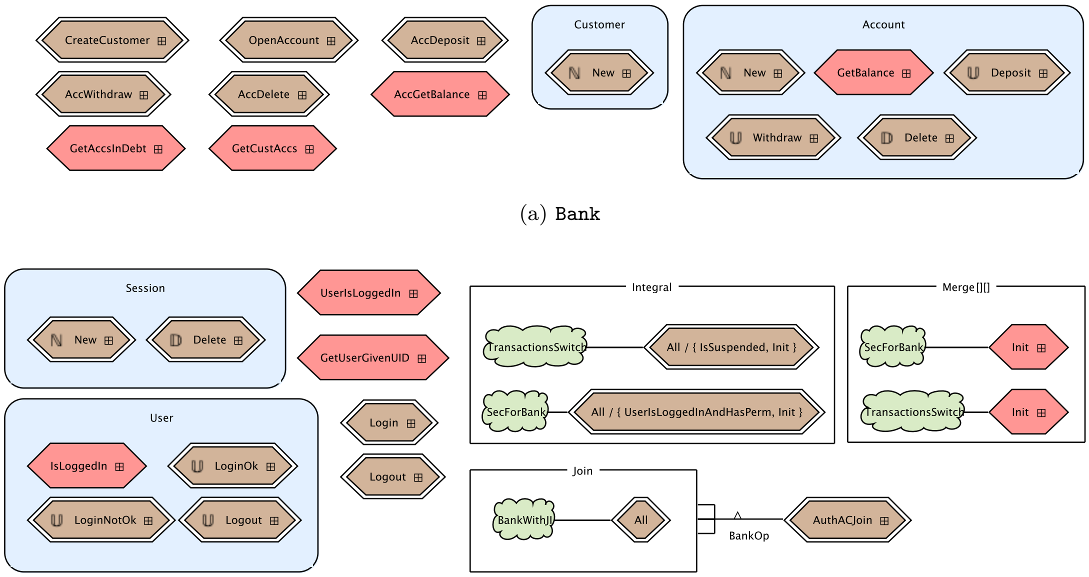
- *Reference* sets include symbol ↑; they are used to refer to sets defined in used packages as defined in the PD (those sets are visible). SDs of Figs. 1.2b and 1.2c have reference sets referring to `Name` and `TimeNat` defined in package `CommonTypes` (alias `CT`).

- 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 or

[2]This defines time as set of time points that are isomorphic to the natural numbers.



(a) `Bank`

(b) `Authentication`

(c) `SecBank`

Figure 1.3: Sample behaviour diagrams of *secure simple bank* (from [Amá11])

blobs. When placed inside sets, constants objects name values or objects of the enclosing set; this is a common idiom to define enumeration sets in VCL (e.g. sets `CustType`, `AccType`, `LoginResult` and `UserStatus` of the SDs of Fig. 1.2 are defined this way). A SD can also include constants with a type designator; such constants are *local* when attached to some set and *global* when they stands alone. In Fig. 1.2c, `maxPwMisses` defines a scalar constant of type `Nat` that is local to set `User`.

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 *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 `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 `TransactionsSwitch` with the exception of `IsSuspended` and `Init` are to be integrally extended; likewise for all operations of package `SecForBank` except `UserIsLoggedInAndHasPerm` and `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 *logical conjunction*. BD of Fig. 1.3c includes a merge box saying that `Init` of `SecForBank` is to be merged with `Init` of `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 *join contracts*, comprising a pre- and post-condition, that is connected to the box through a *fork edge*. There are two kinds of fork edges: *concurrent* (symbol $\wedge$) and *sequential* (symbol ⊡). BD of Fig. 1.3c includes a join extension, saying that all operations of `BankWithJI` are to be concurrently joined with contract `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 `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 ⊞). 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. 1.4c has two assertion imports: one total and one partial. Total assertion import says that assertion `GetBalance` is to be called on `Account` object `a!`; as import is total, variable

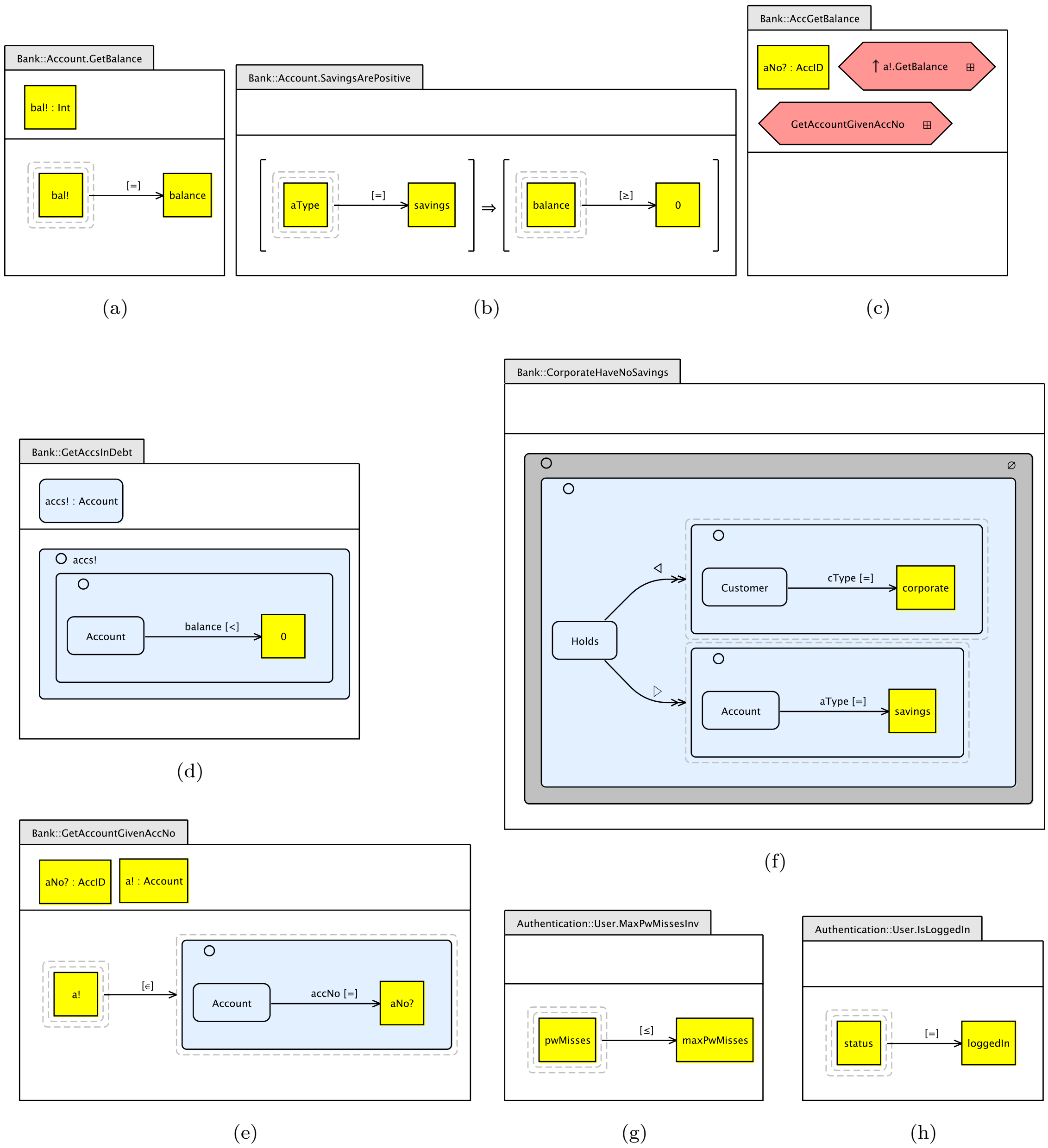


Figure 1.4: Sample ADs of packages of secure simple bank (from [Amá11])

bal! defined in `Account.GetBalance` (Fig. 1.4a) is also defined in `AccGetBalance`. Partial import of `GetAccountGivenAccNo` means that output *a*! 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 and propositional logic operators. Figure 1.4a expresses an equality predicate using a predicate property edge to say that `bal!` is to hold value `Account.balance`. Figure 1.4b expresses an implication formula to say that if the property `aType` of `Account` has value `savings` then the property `balance` must be greater or equal than 0. Figure 1.4d outputs the set of accounts with negative balances; this builds a set using a set definition construction (symbol ◯) by constraining set `Account` using predicate property edges (arrows); the constructed set is then assigned to output 'accs!'. Figure 1.4e expresses a set membership predicate using a predicate property edge to say that output `a!` belongs to accounts with property `accNo` equal to `aNo?` (there is at least one). Figure 1.4f comprises a set formula that defines a set by constraining the relation `Holds`, using property edge modifiers with operators domain ($\lhd$) and range restriction ($\rhd$), to give the set of tuples made of corporate customers and savings accounts; outer shading (reinforced with symbol $\varnothing$) 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. 1.5e, import of contract `HasSessionAddNew` includes a renaming. In Fig. 1.5f, the two imported contracts are combined using a disjunction to say that a login operation is either successful (`LoginOk`, Fig. 1.5e) or not (`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 1.5c, 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.

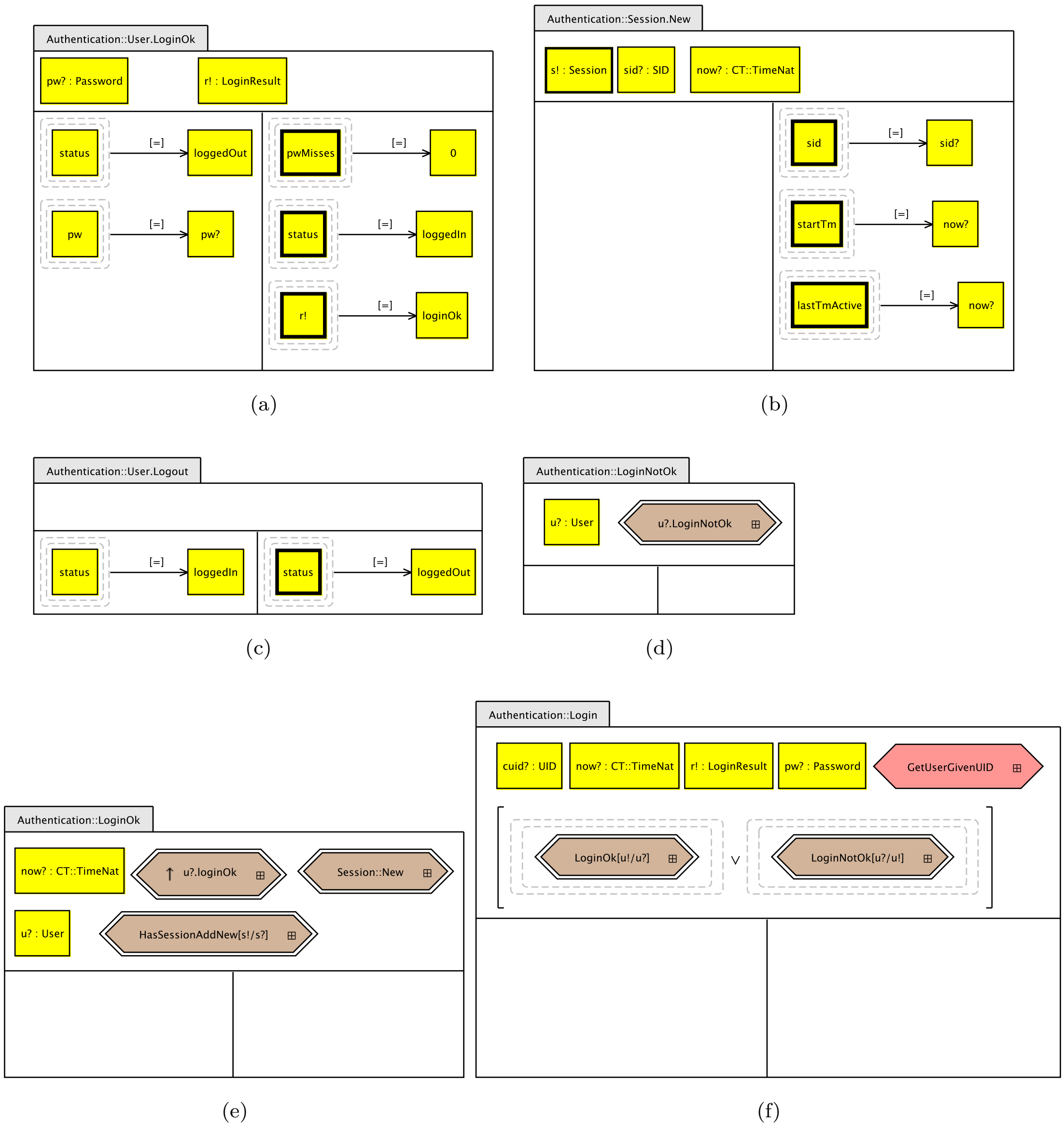


Figure 1.5: Sample CDs of package `Authentication` of secure simple bank (from [Amá11])

- 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.

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 [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

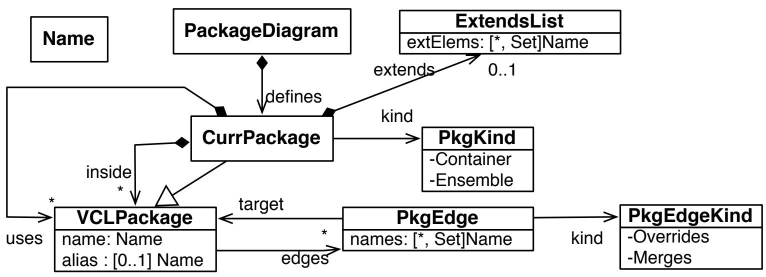


Figure 2.1: The metamodel of VCL Package diagrams

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:

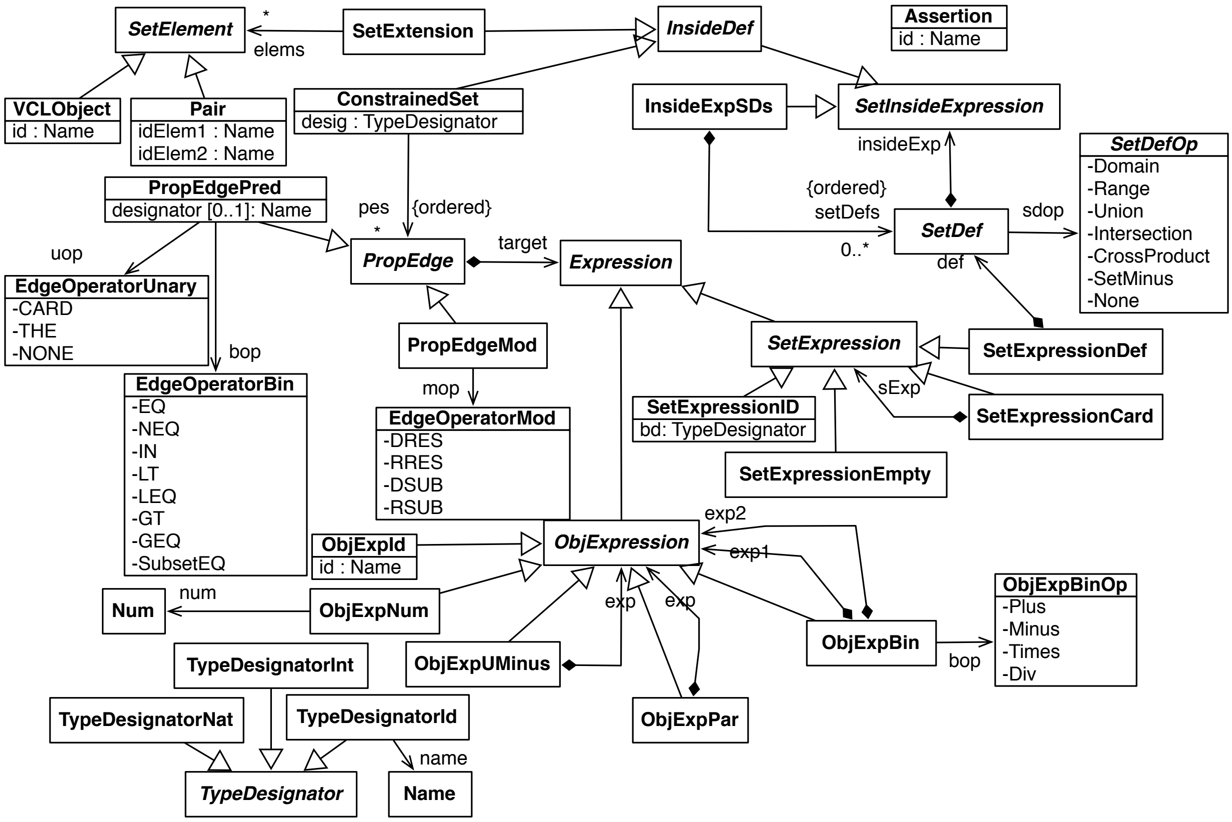


Figure 2.2: The common metamodel

- 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 incorported 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.

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`). `PropEdgePred`s comprise a unary and binary operator (`uop` and `bop` association-ends), an instance of `EdgeOperatorUn` and `EdgeOperatorBin`, respectivelly, a target `Expression` (`target` association-end) and an optional designator (attribute `designator`) 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`, $\lhd$), range restriction (`RRES`, $\rhd$), domain subtraction (`DSUB`, ⩤) and range subtraction (`RSUB`, ⩥). A predicate edge operator is enumeration comprising the operators: equality (`EQ`, $=$), non-equality (`NEQ`, $\neq$), set membership (`IN`, $\in$), less then (`LT`, $<$), less or equal then (`LEQ`, $\leq$), greater then (`GT`, $>$), greater or equal then (`GEQ`, $\geq$), and subsetting (`SubsetEQ`, $\subseteq$).

- 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` (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 $\leftarrow$), `Range` (symbol $\rightarrow$), `Union` (symbol $\cup$), `Intersection` (symbol $\cap$), `CrossProduct` (symbol $\times$), `SetMinus` (symbol $\setminus$) 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

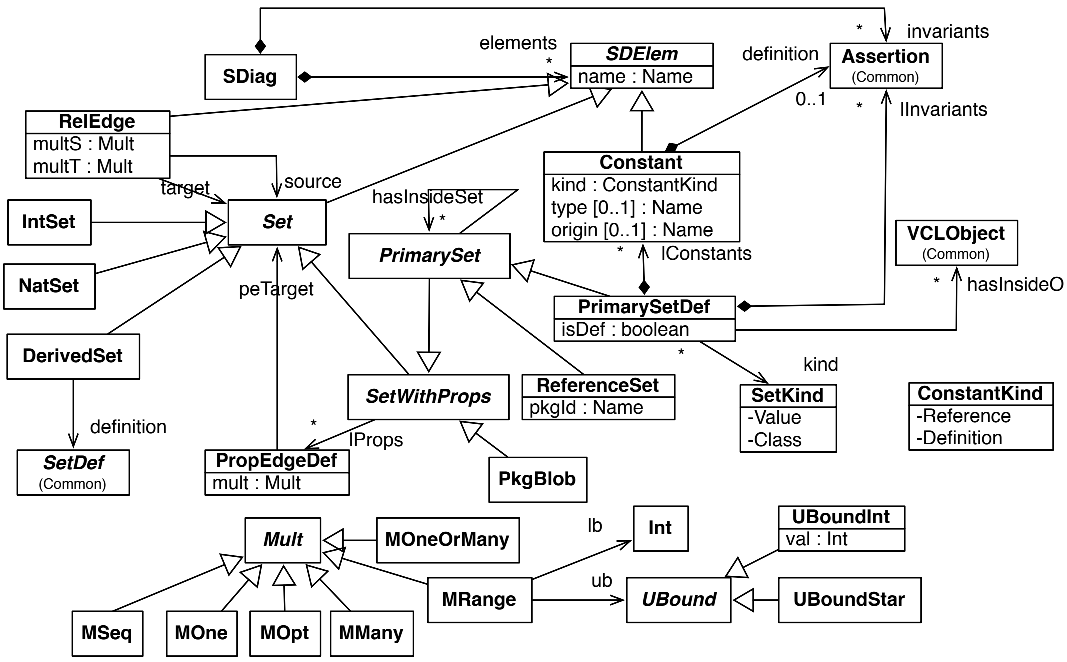


Figure 2.3: The metamodel of VCL Structural diagrams

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`).

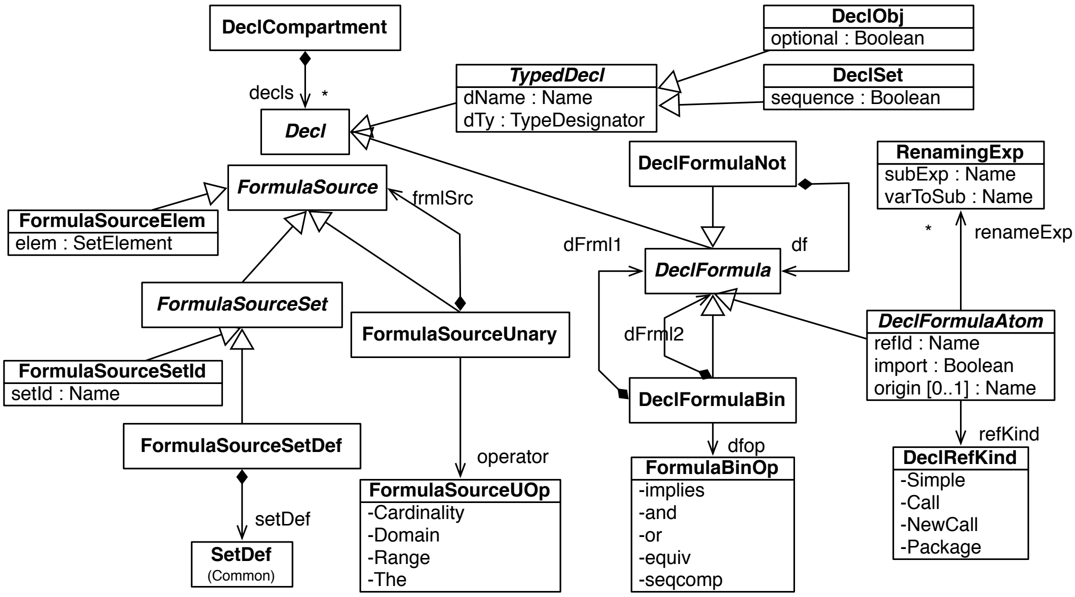


Figure 2.4: The common metamodel of VCL assertion and contract diagrams

- 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:

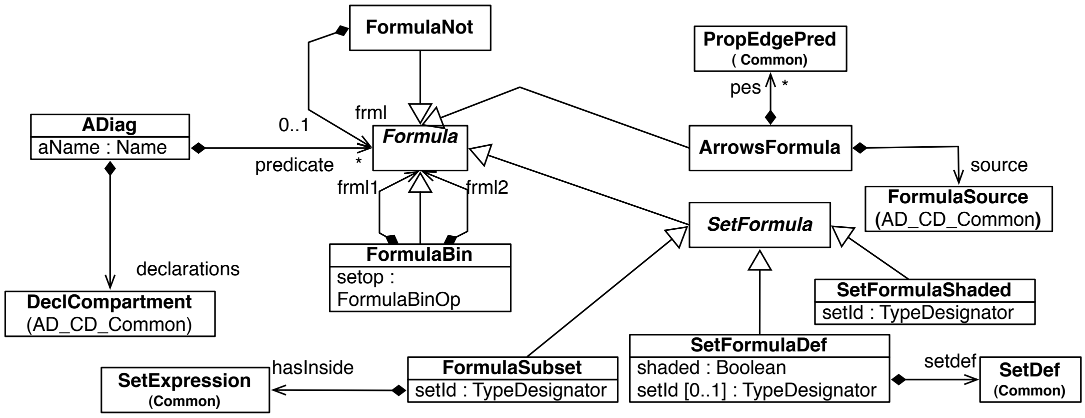


Figure 2.5: The metamodel of VCL assertion diagrams

- 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),

$$\begin{array}{rcl} \mathsf{VCL} & ::= & \overline{Pkg} \\ \mathsf{Pkg} & ::= & \mathsf{PD}\ \mathsf{SD}\ \overline{AD} \end{array}$$

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:

- $\overline{x}$ for zero or more repetitions of x;
- $\overline{x}^1$ for one or more repetitions of x;
- $x \mid y$ for a choice of x or y;
- $[x]$ for an optional x.

In addition,

- $\overline{xc}$ for some character symbol c means zero or more occurrences of x separated with c;
- $\overline{xc}^1$ 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).

$$
\begin{array}{rcl}
\mathsf{PK} & ::= & \textbf{container} \mid \textbf{ensemble} \\
\mathsf{PD} & ::= & \mathsf{PK}\ \textbf{package}\ \mathsf{Id}\ [\textbf{uses}\ \overline{PRef,}] \\
 & & [\textbf{incorporates}\ \overline{PRef,}]\ \pmb{\{}\ \overline{PDep}\ [\mathsf{PExts}]\ \pmb{\}} \\
\mathsf{PRef} & ::= & \mathsf{Id}\ [\textbf{as}\ \mathsf{Id}] \\
\mathsf{PExts} & ::= & \textbf{extends}\ \pmb{(}\overline{Id,}^{1}\pmb{)} \\
\mathsf{PDep} & ::= & \mathsf{Id}\ \mathsf{PEdgKind}\ \mathsf{Id}\ \textbf{on}\ \pmb{(}\ \overline{Id,}^{1}\pmb{)} \\
\mathsf{PEdgKind} & ::= & \textbf{merges} \mid \textbf{overrides}
\end{array}
$$

Figure 2.7: Syntax of Package Diagrams

$$
\begin{array}{rcl}
\mathsf{TD} & ::= & \textbf{Int} \mid \textbf{Nat} \mid [\mathsf{Id}\ \textbf{::}]\ \mathsf{Id} \\
\mathsf{O} & ::= & \textbf{object}\ \mathsf{Id} \\
\mathsf{P} & ::= & \textbf{pair}\ \pmb{(}\mathsf{Id},\ \mathsf{Id}\pmb{)} \\
\mathsf{SE} & ::= & \mathsf{O} \mid \mathsf{P} \\
\mathsf{A} & ::= & \textbf{assertion}\ \ \mathsf{AId} \\
\mathsf{PE} & ::= & (\mathsf{PEP} \mid \mathsf{PEM})\ \mathsf{TExp} \\
\mathsf{PEP} & ::= & \pmb{[}UEOp\pmb{]}\ [\mathsf{Id.}] \rightarrow \pmb{[}BEOp\pmb{]} \\
\mathsf{UEOp} & ::= & \# \mid \odot \mid \perp \\
\mathsf{BEOp} & ::= & \pmb{=} \mid \neq \mid \in \mid < \mid \leq \mid > \mid \geq \mid \subseteq \\
\mathsf{PEM} & ::= & \pmb{[}\ \mathsf{MOp}\ \pmb{]} \rightrightarrows \\
\mathsf{MOp} & ::= & \lhd \mid \rhd \mid \text{⌫} \mid \text{⌦} \\
\mathsf{TExp} & ::= & \textbf{object}\ \pmb{[}\mathsf{OExp}\pmb{]} \mid \mathsf{SExp} \\
\mathsf{OExp} & ::= & \mathsf{Id} \mid \mathsf{Num} \mid -\mathsf{OExp} \mid \mathsf{OExp}\ \mathsf{OEOP}\ \mathsf{OExp} \mid (\mathsf{OExp}) \\
\mathsf{OEOp} & ::= & + \mid - \mid * \mid \textbf{div} \\
\mathsf{SExp} & ::= & \textbf{set}\ \mathsf{TD} \mid \mathsf{SDef} \mid \textbf{set shaded} \mid \#\ \mathsf{SExp} \\
\mathsf{SDef} & ::= & \textbf{set}\ \bigcirc\ \mathsf{SOp}\ \textbf{hasIn}\ \pmb{\{}\mathsf{IExp}\pmb{\}} \\
\mathsf{SOp} & ::= & \leftarrow \mid \rightarrow \mid \cap \mid \cup \mid \times \mid \setminus \mid \perp \\
\mathsf{IExp} & ::= & \mathsf{IDef} \mid \overline{SDef;} \\
\mathsf{IDef} & ::= & \textbf{set}\ \mathsf{TD}\ \pmb{\{}\ \overline{PE}^{1}\ \pmb{\}} \mid \overline{SE}^{1}
\end{array}
$$

Figure 2.8: Common syntactic constructions to ADs and SDs

$$
\begin{array}{rcl}
\mathsf{SD} & ::= & \overline{SDE}\ \overline{A} \\
\mathsf{SDE} & ::= & \mathsf{GC} \mid \mathsf{RE} \mid \mathsf{Set} \mid \mathsf{PkgE} \\
\mathsf{GC} & ::= & \mathsf{C} \mid \mathsf{CR} \\
\mathsf{C} & ::= & \textbf{const}\ \mathsf{Id}\ \textbf{:}\ \mathsf{TD} \mid \textbf{const}\ \mathsf{Id}\ \textbf{:}\ \mathsf{TD} \leftrightarrow \mathsf{A} \\
\mathsf{CR} & ::= & \uparrow \textbf{const}\ [\mathsf{Id}\textbf{.}]\ \mathsf{Id} \leftrightarrow \mathsf{Assertion} \\
\mathsf{M} & ::= & \textbf{opt} \mid \textbf{one} \mid \textbf{some} \mid \textbf{many} \mid \textbf{seq} \mid \mathsf{Num}\ \textbf{..}\ (\mathsf{Num} \mid \textbf{*}) \\
\mathsf{RE} & ::= & \textbf{relEdge}\ \mathsf{Id}\ \textbf{(}\mathsf{M\ TD,\ M\ TD}\textbf{)} \\
\mathsf{SK} & ::= & \textbf{value} \mid \textbf{class} \\
\mathsf{Set} & ::= & \mathsf{PSet} \mid \mathsf{RSet} \mid \mathsf{Id} \leftrightarrow \textbf{set}\ \mathsf{SetDef} \\
\mathsf{RSet} & ::= & \uparrow \textbf{set}\ [\mathsf{Id}\ \textbf{::}]\ \mathsf{Id} \\
\mathsf{PSet} & ::= & \mathsf{PSetDef} \mid \mathsf{ExtSet} \\
\mathsf{PSetDef} & ::= & \textbf{set}\ \mathsf{Id\ SK}\ [\bigcirc]\ \textbf{\{}\ \overline{C}\ \overline{PED}\ \overline{A}\ \textbf{\}}\ [\textbf{hasIn}\ \textbf{\{}\overline{(O \mid PSet)}\textbf{\}}] \\
\mathsf{ExtSet} & ::= & \downarrow \textbf{set}\ \mathsf{Id}\ \textbf{\{}\ \overline{C}\ \overline{PED}\ \overline{A}\ \textbf{\}}\ [\textbf{hasIn}\ \textbf{\{}\overline{(O \mid PSet)}\textbf{\}}] \\
\mathsf{PkgE} & ::= & \textbf{pkg}\ \mathsf{Id}\ \textbf{\{}\ \overline{PED}\ \textbf{\}} \\
\mathsf{PED} & ::= & \mathsf{Id} \rightarrow \mathsf{M\ TD}
\end{array}
$$

Figure 2.9: Syntax of structural diagrams

$$
\begin{array}{rcl}
\mathsf{AD} & ::= & \textbf{AD}\ \mathsf{Id}\ [\mathsf{:Id}]\ \textbf{decls}\ \textbf{\{}\overline{D}\textbf{\}}\ \textbf{pred}\ \textbf{\{}\overline{F}\textbf{\}} \\
\mathsf{D} & ::= & \mathsf{DV\ Id}\ \textbf{:}\ \mathsf{TD} \mid \mathsf{DF} \\
\mathsf{DV} & ::= & \textbf{object}\ [\textbf{?}] \mid \textbf{set :}\ [\textbf{[]}] \\
\mathsf{R} & ::= & \mathsf{Id}\ \textbf{/}\ \mathsf{Id} \\
\mathsf{DFA} & ::= & [\uparrow]\ \mathsf{A}\ \textbf{[}\overline{R}\textbf{]} \mid [\uparrow]\ \textbf{assertion}\ \mathsf{Id}\ (\rightarrow \mid \textbf{.} \mid \textbf{::})\ \mathsf{Id}\ \textbf{[}\overline{R}\textbf{]} \\
\mathsf{DF} & ::= & \mathsf{DFA} \mid \neg\ \textbf{(}\ \mathsf{DF}\ \textbf{)} \mid \textbf{(}\ \mathsf{DF\ FOp\ DF}\ \textbf{)} \\
\mathsf{FOp} & ::= & \Rightarrow \mid \Leftrightarrow \mid \wedge \mid \vee \mid \boxdot \\
\mathsf{F} & ::= & \mathsf{AF} \mid \mathsf{SF} \mid \neg\ \textbf{[F]} \mid \textbf{[F]}\ \mathsf{FOp}\ \textbf{[F]} \\
\mathsf{AFS} & ::= & \mathsf{SE} \mid \mathsf{AFSS} \mid \mathsf{FSOp\ AFS} \\
\mathsf{AFSS} & ::= & \textbf{set}\ \mathsf{Id} \mid \mathsf{SDef} \\
\mathsf{FSOp} & ::= & \# \mid \leftarrow \mid \rightarrow \mid \odot \\
\mathsf{AF} & ::= & \mathsf{AFS}\ \textbf{\{}\ \overline{PEP}\ \textbf{\}} \\
\mathsf{SF} & ::= & [\textbf{shaded}]\ [\mathsf{Id}]\ \mathsf{SDef} \mid \textbf{set shaded}\ \mathsf{TD} \mid \textbf{set}\ \mathsf{TD}\ \textbf{hasIn}\ \textbf{\{}\mathsf{SExp}\textbf{\}}
\end{array}
$$

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:

$$VE == Id \nrightarrow T$$

VCL's types (set T) are as follows:

$$\begin{aligned}\mathsf{T} \quad ::= \quad & \textbf{Int} \mid \textbf{Nat} \mid \textbf{Null} \mid \textbf{Set}\ \mathsf{T} \mid \textbf{Seq}\ \mathsf{T} \mid \textbf{Opt}\ \mathsf{T} \mid \textbf{Top} \mid \textbf{Obj} \mid \textbf{Set}\ \mathsf{Id} \mid \textbf{Pair}\ \textbf{(}\mathsf{T}, \mathsf{T}\textbf{)} \\ & \mid \textbf{Assertion}\ \textbf{[}VE_v,\ VE_h\textbf{]} \mid \textbf{Contract}\ \textbf{[}\ VE_v,\ VE_h\ \textbf{]} \mid \textbf{Pkg}\ \mathsf{Id}\end{aligned}$$

Here, (a) `Int` represents the integers, (b) `Nat` the natural numbers; (c) `Null` represents erroneous results (implementation only); (d) **Set** `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; (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 owning 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

$E \vdash T$	T is well-formed type in E
$E \vdash T_1 <: T_2$	T_1 is a subtype of T_2 in E
$E \vdash Id : T$	Id is well-formed identifier of type T in E
$E \vdash Id_s.Id_l : T$	Id_l is well-formed local identifier of set Id_s with type T in E
$E \vdash Id_p::Id : T$	Id is well-formed identifier of package Id_p with type T in E

$$
\begin{aligned}
&SK ::= \textbf{value} \mid \textbf{class}\\
&DK ::= \textbf{def} \mid \textbf{notDef}\\
&PE == Id \nrightarrow PK \times E\\
&SE == Id \nrightarrow SK \times DK \times Id \times VE\\
&SubE == T \leftrightarrow T\\
&E == VE \times PE \times SE \times SubE
\end{aligned}
$$

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.

$$DE == Id \nrightarrow Pkg$$

We introduce the following conventions:

- $\overline{X}$ is a sequence of some set X.
- $E_\varnothing$ is an empty environment. $E.VE$, $E.PE$, $E.SE$ and $E.SubE$ denote the different components of E.
- $Id : T$, $Id \overset{se}{\mapsto} (SK, DK, Id, VE)$ and $Id \overset{pe}{\mapsto} (PK, E)$ 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, 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 `Ty Id` 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

$$(\mathit{Ty\ Id})\quad \frac{(\mathit{Id},\, T) \in E.\mathit{VE}}{E \vdash \mathit{Id} : T}$$

$$(\text{Type})\quad \frac{\begin{array}{c} T = \mathbf{Set}\,\mathit{Id} \Rightarrow \mathit{Id} \in \operatorname{dom} E.\mathit{SE} \\ T = \mathbf{Pkg}\,\mathit{Id} \Rightarrow \mathit{Id} \in \operatorname{dom} E.\mathit{PE} \end{array}}{E \vdash T}$$

$$(\text{SetId Prop})\quad \frac{\begin{array}{c} E \vdash \mathbf{Set}\,\mathit{Id}_s \qquad \mathit{Id}_l \in \operatorname{dom}(E.\mathit{SE}\,(\mathit{Id}_s)).\,\mathit{VE} \\ E \oplus (E.\mathit{SE}\,(\mathit{Id}_s)).\,\mathit{VE} \vdash \mathit{Id}_l : T \end{array}}{E \vdash \mathit{Id}_s.\mathit{Id}_l : T}$$

$$(\mathit{Package\ Id})\quad \frac{E \vdash \mathit{Id} : \mathbf{Pkg}\,\mathit{Id}_p \qquad (E.\mathit{PE}(\mathit{Id}_p)).E \vdash \mathit{Id} : T}{E \vdash \mathit{Id}_p\mathbf{::}\mathit{Id} : T}$$

Table 3.3 Basic VCL sub-typing rules

$$(\text{SubTy})\quad \frac{\begin{array}{c} E \vdash T_1 \qquad E \vdash T_2 \\ (T_1,\, T_2) \in E.\mathit{SubE} \end{array}}{E \vdash T_1 <: T_2} \qquad (\mathit{Sub\ Refl})\quad \frac{E \vdash T}{E \vdash T <: T}$$

$$(\text{Sub Trans})\quad \frac{\begin{array}{c} E \vdash T_A <: T_B \\ E \vdash T_B <: T_C \end{array}}{E \vdash T_A <: T_C} \qquad (\text{Subsumption})\quad \frac{\begin{array}{c} E \vdash I : T_A \\ E \vdash T_A <: T_B \end{array}}{E \vdash I : T_B} \qquad (\mathit{Sub\ Top})\quad \frac{E \vdash T}{E \vdash T <: \mathit{Top}}$$

$$(\mathit{Sub\ Obj})\quad \frac{E \vdash \mathbf{Set}\,\mathit{Id}_s}{E \vdash \mathbf{Set}\,\mathit{Id}_s <: \mathit{Obj}} \qquad (\mathit{Sub\ NatInt})\quad \frac{}{E \vdash \mathit{Nat} <: \mathit{Int}}$$

$$(\mathit{Sub\ Pow})\quad \frac{E \vdash T_A <: T_B}{E \vdash \mathbf{Pow}\,T_A <: \mathbf{Pow}\,T_B} \qquad (\mathit{Sub\ Seq})\quad \frac{E \vdash T_A <: T_B}{E \vdash \mathbf{Seq}\,T_A <: \mathbf{Seq}\,T_B}$$

$$(\mathit{Sub\ Opt})\quad \frac{E \vdash T_A <: T_B}{E \vdash \mathbf{Opt}\ T_A <: \mathbf{Opt}\ T_B} \qquad (\mathit{Sub\ Opt\ PSet})\quad \frac{E \vdash T_A <: T_B}{E \vdash \mathbf{Opt}\ T_A <: \mathbf{Pow}\ T_B}$$

$$(\mathit{Sub\ Pair})\quad \frac{E \vdash T_{A1} <: T_{A2} \qquad E \vdash T_{B1} <: T_{B2}}{E \vdash \mathbf{Pair}\,(T_{A1},\, T_{B1}) <: \mathbf{Pair}\,(T_{A2},\, T_{B2})}$$

are valid provided their identifiers are defined in set and package environments; all remaining types are valid. Rule `SetId Prop` yields the type that is associated with some local identifier Id_l of some set Id_s; 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 `Package Id` retrieves the type of some identifier Id defined in some package Id_p; this requires that there is a package type defined for the given package identifier Id_p and that the given identifier is defined in the package's environment.

Table 3.3 lists basic subtyping rules. Rule `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 ($E.\mathit{SubE}$). Rules `Sub Refl` and `Sub Trans` says that the subtyping relation is both reflexive and transitive. Rule `Subsumption` 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 `Sub Top` says that any valid type is a subtype of type `Top`. Rule `Sub Obj` says that any set type is a subtype of type `Obj`. Rule `Sub NatInt` says that type of natural numbers is a subtype of the integers. Rules `SubPow`, (Sub Seq) and `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 `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 `Sub Pair` says that two pair types are subtypes of each other if their corresponding components are subtypes of each other also.

Table 3.4 Judgements for common syntactic constructions of VCL ADs and SDs

$E \vdash^{td} TD \ : \ T$	TD is well-formed type designator with type T in E
$E \vdash^{ta} A \therefore AId : T$	A is well-formed assertion with identifier AId and type T in E
$E \vdash^{se} SE : T$	SE is well-formed set element with type T in E
$E \vdash^{sdef} SDef \ : T$	Set definition $SDef$ yields type T in E
$E \vdash^{id} IDef : T$	Inside definition $IDef$ yields type T in E
$E; SOP \vdash^{sdo} \overline{T} : T$	$\overline{T}$ is sequence of types yielding type T when applied to operator SOp
$E; T \vdash^{pe} PE$	Property edge PE is well-formed in E
$E; T \vdash^{pep} PEP$	Predicate property edge PEP is well-formed in E
$E; T \vdash^{pem} PEM$	Modifier property edge PEM is well-formed in E
$E \vdash^{te} TExp : T$	Target expression $TExp$ yields type T in E
$E \vdash^{oe} OExp : T$	Object expression $OExp$ yields type T in E
$E \vdash^{sexp} SExp : T$	Set expression $SExp$ yields type T in E
$E; UEOp \vdash^{ueop} T_1 : T_2$	Type T_1 yields type T_2 when applied unary edge operator $UEOp$
$E; BEOP \vdash^{eop} (T_1, T_2)$	Types T_1, T_2 are well typed when applied predicate edge operator $BEOP$
$E; MOP \vdash^{mop} (T_1, T_2)$	Types T_1, T_2 are well typed when applied modifier edge operator MOP
$E; \overline{AD}; Id_{s\perp}; Id_{c\perp} \vdash^{aok} A \therefore VE$	Assertion A has AD yielding variable environment VE
$E; \overline{AD}; Id_{s\perp} \vdash^{adok} \overset{\frown}{AD} \therefore VE$	$\overset{\frown}{AD}$ is set of ADs yielding variable environment VE

Table 3.5 Type rules for type designators (TD non-terminal)

$$(TD\ Nat)\ \frac{}{E \vdash^{td} \mathbf{Nat} \ : \ Nat} \qquad (TD\ Int)\ \frac{}{E \vdash^{td} \mathbf{Int} \ : \ Int} \qquad (TD\ Id)\ \frac{E \vdash Id \ : \ T}{E \vdash^{td} Id \ : \ T} \qquad (TD\ Id\ Pkg)\ \frac{E \vdash Id_p :: Id \ : \ T}{E \vdash^{td} Id_p :: Id \ : \ T}$$

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 3.5) 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 3.6 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` $\overline{SDef}$). The rule essentially derive a sequence of types from inside definition ($IDef$) or sequence of set definitions ($\overline{SDef}$) 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

Table 3.6 Type rules for assertions and set elements

(T Assertion)
$$\frac{E \vdash Id : T \quad T = \mathbf{Assertion}[VE_v, VE_h]}{E \vdash^{ta} \mathbf{assertion}\, Id \,\therefore\, Id \,:\, T}$$

$(T\ SE\ Obj)$
$$\frac{E \vdash Id : T \qquad T <: \mathbf{Obj}}{E \vdash^{se} \mathbf{object}\, Id \,\therefore\, Id \,:\, T}$$

$(T\ SE\ Pair)$
$$\frac{E \vdash Id_1 : T_1 \qquad E \vdash Id_2 : T_2}{E \vdash^{se} \mathbf{pair}(Id_1, Id_2) : \mathbf{Pair}\,(T_1, T_2)}$$

Table 3.7 Type rules for set definitions and associated inside definitions

$(T\ SDef\ IDef)$
$$\frac{E \vdash^{id} IDef : T_1 \qquad E; SOp \vdash^{sdo} T_1 : T}{E \vdash^{sdef} \mathbf{set} \bigcirc SOp\, \mathbf{hasIn}\, \{IDef\} : T}$$

$(T\ SDef\ \overline{SDef})$
$$\frac{E \vdash^{sdef} \overline{SDef};: \overline{T_{sd}} \qquad E; SOp \vdash^{sdo} \overline{T_{sd}} : T_f}{E \vdash^{sdef} \mathbf{set} \bigcirc SOp\, \mathbf{hasIn}\, \{\overline{SDef};\} : T_f}$$

(IDef CntSet)
$$\frac{E \vdash^{td} TD : T \quad E; T \vdash^{pe} \overline{PE} \quad (IsSeqOfPEP(\overline{PE}) \vee IsSeqOfPEM(\overline{PE}))}{E \vdash^{id} \mathbf{set}\ TD\, \{\overline{PE}\} : \mathbf{Pow}\, T}$$

$(IDef\ SE)$
$$\frac{E \vdash^{se} SE : T}{E \vdash^{id} SE : \mathbf{Pow}\, T}$$

$(IDef\ \overline{SE} *)$
$$\frac{E \vdash^{se} SE : T_1 \qquad E \vdash^{id} \overline{SE} : T_2 \quad (T_r = T_1 \wedge T_2 <: T_1) \vee (T_r = T_2 \wedge T_1 <: T_2)}{E \vdash^{id} SE\ \overline{SE} : \mathbf{Pow}\, T_r}$$

modifier (disjunction). The type rules for set extensions (`IDef SE` and `IDef SE *`) process the sequence 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 (`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 `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 (`SOp Dom` and `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 `SOp Dom` returns a type formed as the powerset of the first type of the pair (the domain). Rule `SOp Ran` returns a type formed as the powerset of the second type (the range).
- The cross product rules (`SOp Cross` and `SOp * 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 powerset types and yields a powerset of a pair type. Rule `SOp * 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 (`SOp Pair Intersection` and `SOp * Intersection`) and union rules (`SOp Pair Union` and `SOp * 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 *getGType*, appendix A). All given types must be subtypes of each other. The set subtraction rule (`SOp Pair SetMinus`) does the same for a pair of powerset types.

Table 3.8 Type rules for set def operators

(*SOp None*)

$$\frac{}{E; \perp \vdash^{sdo} T : T}$$

(*SOp Dom*)

$$\frac{}{E; \leftarrow \vdash^{sdo} \mathbf{Pow}\,\mathbf{Pair}\,(T_d, T_r) : \mathbf{Pow}\ T_d}$$

(*SOp Ran*)

$$\frac{}{E; \rightarrow \vdash^{sdo} \mathbf{Pow}\,\mathbf{Pair}\,(T_d, T_r) : \mathbf{Pow}\ T_r}$$

(*SOp Cross*)

$$\frac{}{E; \times \vdash^{sdo} \mathbf{Pow}\ T_1\ \mathbf{Pow}\ T_2 : \mathbf{Pow}\,\mathbf{Pair}\,(T_1, T_2)}$$

(*SOp* $*$ *Cross*)

$$\frac{E; \times \vdash^{sdo} \overline{T}^1 : \mathbf{Pow}\ T_2}{E; \times \vdash^{sdo} \mathbf{Pow}\ T_1\ \overline{T}^1 : \mathbf{Pow}\,\mathbf{Pair}(T_1, T_2)}$$

(*SOp Pair Intersection*)

$$\frac{T = getGType(E, T_1, T_2)}{E; \cap \vdash^{sdo} \mathbf{Pow}\ T_1\ \mathbf{Pow}\ T_2 : \mathbf{Pow}\ T}$$

(*SOp* $*$ *Intersection*)

$$\frac{E; \cap \vdash^{sdo} \overline{T}^1 : \mathbf{Pow}\ T_2 \qquad T = getGType(E, T_1, T_2)}{E; \cap \vdash^{sdo} \mathbf{Pow}\ T_1\ \overline{T}^1 : \mathbf{Pow}\ T}$$

(*SOp Pair Union*)

$$\frac{T = getGType(E, T_1, T_2)}{E; \cup \vdash^{sdo} \mathbf{Pow}\ T_1\ \mathbf{Pow}\ T_2 : \mathbf{Pow}\ T}$$

(*SOp* $*$ *Union*)

$$\frac{E; \cup \vdash^{sdo} \overline{T}^1 : \mathbf{Pow}\ T_2 \qquad T = getGType(E, T_1, T_2)}{E; \cup \vdash^{sdo} \mathbf{Pow}\ T_1\ \overline{T}^1 : \mathbf{Pow}\ T}$$

(*SOp Pair SetMinus*)

$$\frac{T = getGType(E, T_1, T_2)}{E; \setminus \vdash^{sdo} \mathbf{Pow}\ T_1\ \mathbf{Pow}\ T_2 : \mathbf{Pow}\ T}$$

Table 3.9 Type rules for property edges

(PE PEP)

$$\frac{E \vdash^{te} TExp : T_2 \qquad E; (T_1, T_2) \vdash^{pep} PEP}{E; T_1 \vdash^{pe} PEP\ TExp}$$

(PE PEM)

$$\frac{E \vdash^{te} TExp : T_2 \qquad E; (T_1, T_2) \vdash^{pem} PEM}{E; T_1 \vdash^{pe} PEM\ TExp}$$

(PE Many)

$$\frac{E; T \vdash^{pe} PEP \qquad E; T \vdash^{pe} \overline{PEP}^1}{E; T \vdash^{pe} PEP\ \overline{PEP}^1}$$

(PEP)

$$\frac{E; T_1 \vdash^{peps} [Id] : T_s \qquad E; UEOp \vdash^{ueop} T_s : T_{sf} \qquad E; BEOP \vdash^{eop} (T_{sf}, T_2)}{E; (T_1, T_2) \vdash^{pep} [UEOp]\,[Id] \rightarrow [BEOP]}$$

(*PEPS* ϵ)

$$\frac{}{E; T \vdash^{peps} \epsilon : T}$$

(*PEPS PId*)

$$\frac{T = \mathbf{Set}\ Id_s \qquad E \vdash Id_s.Id_{pr} : T_p}{E; T \vdash^{peps} Id_{pr} : T_p}$$

(*PEPM*)

$$\frac{E; MOp \vdash^{mop} (T_1, T_2)}{E; (T_1, T_2) \vdash^{pem} [MOp]\ \rightrightarrows}$$

(*UEOp No*)

$$\frac{}{E; \perp \vdash^{ueop} T : T}$$

(*UEOp Card*)

$$\frac{}{E; \# \vdash^{ueop} \mathbf{Pow}\ T : \mathbf{Int}}$$

(*UEOp The*)

$$\frac{}{E; \odot \vdash^{ueop} \mathbf{Opt}\ T : T}$$

Table 3.10 Type rules for binary predicate edge operators (BEOp)

$$(\mathit{BEOP\ EQ})\quad \frac{(E \vdash T_1 <: T_2 \vee E \vdash T_2 <: T_1)}{E; = \vdash^{eop} (T_1, T_2)}$$

$$(\mathit{BEOP\ NEQ})\quad \frac{E; = \vdash (T_1, T_2)}{E; \neq \vdash^{eop} (T_1, T_2)}$$

$$(\mathit{BEOP\ IN})\quad \frac{E \vdash T_1 <: T_2}{E; \in \vdash^{eop} (T_1, \mathbf{Pow}\ T_2)}$$

$$(\mathit{BEOP\ LT})\quad \frac{E \vdash T_1 <: \mathit{Int} \qquad E \vdash T_2 <: \mathit{Int}}{E; < \vdash^{eop} (T_1, T_2)}$$

$$(\mathit{BEOP\ LEQ})\quad \frac{E \vdash T_1 <: \mathit{Int} \qquad E \vdash T_2 <: \mathit{Int}}{E; \leq \vdash^{eop} (T_1, T_2)}$$

$$(\mathit{BEOP\ GT})\quad \frac{E \vdash T_1 <: \mathit{Int} \qquad E \vdash T_2 <: \mathit{Int}}{E; > \vdash^{eop} (T_1, T_2)}$$

$$(\mathit{BEOP\ GEQ})\quad \frac{E \vdash T_1 <: \mathit{Int} \qquad E \vdash T_2 <: \mathit{Int}}{E; \geq \vdash^{eop} (T_1, T_2)}$$

$$(\mathit{BEOP\ SUBSETEQ})\quad \frac{E \vdash T_1 <: T_2}{E; \subseteq \vdash^{eop} (\mathbf{Pow}\ T_1, \mathbf{Pow}\ T_2)}$$

Table 3.11 Type rules for modifier edge operators (EOM)

$$(\text{EOMDRES})\quad \frac{T_1 = \mathbf{Pow}\, T_{I1} \qquad T_2 = \mathbf{Pow\, Pair}\,(T_d, T_r) \qquad E \vdash T_{I1} :< T_d \qquad E \vdash T_1 \qquad E \vdash T_2}{E; \lhd \vdash^{mop} (T_1, T_2)}$$

$$(\text{EOMRRES})\quad \frac{T_1 = \mathbf{Pow}\, T_{I1} \qquad T_2 = \mathbf{Pow\, Pair}\,(T_d, T_r) \qquad E \vdash T_{I1} :< T_r \qquad E \vdash T_1 \qquad E \vdash T_2}{E; \rhd \vdash^{mop} (T_1, T_2)}$$

$$(\text{EOMDSUB})\quad \frac{T_1 = \mathbf{Pow}\, T_{I1} \qquad T_2 = \mathbf{Pow\, Pair}\,(T_d, T_r) \qquad E \vdash T_{I1} :< T_r \qquad E \vdash T_1 \qquad E \vdash T_2}{E; ⩤ \vdash^{mop} (T_1, T_2)}$$

$$(\text{EOMRSUB})\quad \frac{T_1 = \mathbf{Pow}\, T_{I1} \qquad T_2 = \mathbf{Pow\, Pair}\,(T_d, T_r) \qquad E \vdash T_{I1} :< T_r \qquad E \vdash T_1 \qquad E \vdash T_2}{E; ⩥ \vdash^{mop} (T_1, T_2)}$$

Table 3.12 Type rules for target expressions

$$(\mathit{TE\ OExp})\quad \frac{E \vdash^{oe} \mathit{OExp} : T}{E \vdash^{te} \mathbf{object}\,[\mathit{OExp}] : T}$$

$$(\mathit{TE\ SExp})\quad \frac{E \vdash^{sexp} \mathit{SExp} : T}{E \vdash^{te} \mathit{SExp} : T}$$

Table 3.13 Type rules for set expressions

$$(\text{SExp TD})\quad \frac{E \vdash^{td} \mathit{TD} : T}{E \vdash^{sexp} \mathbf{set}\, \mathit{TD} : \mathbf{Pow}\ T}$$

$$(\text{SExp SDef})\quad \frac{E \vdash^{sdef} \mathit{SDef} : T}{E \vdash^{sexp} \mathit{SDef} : T}$$

$$(\text{SExp Empty})\quad \frac{}{E \vdash^{sexp} \mathbf{set\ shaded} : \mathbf{Pow\ Top}}$$

$$(\text{SExp Card})\quad \frac{E \vdash^{sexp} \mathit{SExp} : \mathbf{Pow}\, T}{E \vdash^{sexp} \#\, \mathit{SExp} : \mathbf{Int}}$$

Table 3.14 Type rules for object expressions

$$(\mathit{OExp\ ID})\quad \frac{E \vdash \mathit{Id} : T}{E \vdash^{oe} \mathit{Id} : T} \qquad (\mathit{OExp\ Num})\quad \frac{}{E \vdash^{oe} \mathit{Num} : \mathit{Nat}} \qquad (\mathit{OExp\ Uminus})\quad \frac{E \vdash^{oe} \mathit{OE} : \mathit{Int}}{E \vdash^{oe} -\mathit{OE} : \mathit{Int}}$$

$$(\mathit{OExp\ OEOP})\quad \frac{E \vdash^{oe} \mathit{OE}_1 : \mathit{Int} \qquad E \vdash^{oe} \mathit{OE}_2 : \mathit{Int}}{E \vdash^{oe} \mathit{OE}_1\ \mathit{OEOP}\ \mathit{OE}_2 : \mathit{Int}}$$

Table 3.15 Type rules for checking assertions

$$\text{(Assertion Ok)}\quad \frac{\begin{array}{c} \mathit{Id}_{FA} = \mathit{getFAId}(\mathit{Id}_{c\perp}, \mathit{Id}_A) \\ \mathit{Id}_{FA} \notin \operatorname{dom} E.\mathit{VE} \\ \mathit{AD} = \mathit{findAD}(\overline{\mathit{AD}}, \mathit{Id}_{FA}, \mathit{Id}_{s\perp}) \\ E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp}; \mathit{Id}_{c\perp} \vdash^{adok} \mathit{AD} \therefore \mathit{VE} \end{array}}{E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp}; \mathit{Id}_{c\perp} \vdash^{aok} \textbf{assertion}\ \mathit{Id}_A \therefore \mathit{VE}}$$

$$\text{(AD Ok)}\quad \frac{\begin{array}{c} \widehat{\mathit{AD}} = \mathit{getDepsOfAD}\ (\mathit{AD}, \overline{\mathit{AD}}, \mathit{Id}_{s\perp}) \\ E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp} \vdash^{adok} \widehat{\mathit{AD}} \therefore \mathit{VE} \\ \mathit{Id}_{c\perp} \neq \perp \Rightarrow \widehat{\mathit{AD}} = \{\} \\ E, \mathit{VE} \vdash^{ad} \mathit{AD} \therefore \mathit{Id}_A : T \end{array}}{E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp}; \mathit{Id}_{c\perp} \vdash^{adok} \mathit{AD} \therefore \mathit{VE}, \mathit{Id}_A : T}$$

$$(\mathit{AD\ Ok}\ \epsilon)\quad \frac{}{E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp} \vdash^{adok} \{\} \therefore \mathit{VE}_{\varnothing}} \qquad (\mathit{AD\ Ok} *)\quad \frac{E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp}; \perp \vdash^{adok} \mathit{AD} \therefore \mathit{VE} \qquad E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp} \vdash^{adok} \widehat{\mathit{AD}} \therefore \mathit{VE}'}{E; \overline{\mathit{AD}}; \mathit{Id}_{s\perp} \vdash^{adok} \{\mathit{AD}\} \cup \widehat{\mathit{AD}} \therefore \mathit{VE}, \mathit{VE}'}$$

Table 3.15 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 **`getFAId`**, which considers the special case of assertions associated with constants, and then looks for the AD using function **`findAD`** (both functions defined in appendix A, section A.3.9). The retrieved AD is then checked (rule associated with judgement $\vdash^{adok}$) 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 **`getDepsOfAD`** (appendix A, section A.3.8) to yield set $\widehat{AD}$ and then checks them using the rules associated with judgement $\vdash^{adok}$ to derive 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 $AD\ Ok\ \epsilon$ and $AD\ Ok *$ process a set of ADs inductively. Rule $AD\ Ok\ \epsilon$ 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 3.16. 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

Table 3.16 Typing Judgements for VCL Package Diagrams

$DE \vdash^{pd} PD \therefore (Id_p, PK, E)$	Well-formed PD yields triple (Id_p, PK, E), comprising the package's identifier and kind, and an environment
$DE; E \vdash^{puse} \overline{PRef} \therefore E'$	Well-formed sequence of package uses $\overline{PRef}$ yields environment E'
$E; Id_p \vdash^{pdep} \overline{PDep} \therefore E'$	Well-formed sequence of package dependencies $\overline{PDep}$ yields E'
$E; Id_p \vdash^{pm} (Id_{p1}, Id_{p2}, \overline{Id_s}^1) \therefore E'$	Identifiers declaring package merge Id_{p1}, Id_{p2} and $\overline{Id_s}^1$ yield E'
$E; Id_p \vdash^{po} (Id_{p1}, Id_{p2}, \overline{Id_s}^1) \therefore E'$	Identifiers declaring package overrides Id_{p1}, Id_{p2} and $\overline{Id_s}^1$ yield E'
$E; Id_p \vdash^{pext} PExts \therefore E'$	Well-formed package extends declaration $PExts$ yields E'
$DE; E; PK \vdash^{pinc} \overline{Id_p} \therefore E'$	Sequence of incorporated packages $\overline{Id_p}$ yields E'

Table 3.17 Type rules for package diagrams (production PD)

(Ok PD)

$$\frac{\begin{array}{c} NoClashes(\overline{PRef_1}\ \overline{PRef_2}) \qquad DE; E_{\varnothing} \vdash^{puse} \overline{PRef_1}\ \overline{PRef_2} \therefore E \\ E; Id_p; \overline{PRef_2} \vdash^{pdep} \overline{PDep} \therefore E' \qquad E'; PK \vdash^{pinc} \overline{PRef_2} \therefore E'' \qquad E''; Id_p \vdash^{pext} [PExts] \therefore E_f \end{array}}{DE \vdash^{pd} PK\ \textbf{package}\ Id_p\ [\textbf{uses}\ \overline{PRef_1}]\ [\textbf{incorporates}\ \overline{PRef_2}]\ \{\overline{PDep}\ [PExts]\} \therefore (Id_p,\ PK,\ E_f)}$$

a PD. This comprises package uses (judgement labelled `puse`), package dependencies (`pdep`), package merges (`pm`), package overrides (`po`) and package incorporations (`pinc`).

The type rule for PDs (rule `OK PD`, Table 3.17) checks: (a) that there are no name clashes in imports and incorporates using predicate `NoClashes`(appendix A, section A.2.3), (b) the well-formedness of packages being used ($\overline{PRef_1}$) and incorporated ($\overline{PRef_2}$) to produce an updated environment E, (c) the package dependencies ($\overline{PDep}$) 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 3.18 Type rules for package uses

($PUses *$)

$$\frac{DE; E \vdash^{puse} PRef \therefore E'' \qquad DE; E'' \vdash^{puse} \overline{PRef} \therefore E'}{DE; E \vdash^{puse} PRef\ \overline{PRef} \therefore E'}$$

($Puses\ \epsilon$)

$$\frac{}{DE; E \vdash^{puse} \epsilon \therefore E_{\varnothing}}$$

($Puses\ NoAlias$)

$$\frac{Id_p \in \operatorname{dom} DE \qquad Pkg = DE\ Id_p \qquad \neg\ Id_p \in \operatorname{dom} E \qquad DE \vdash^{pkg} Pkg : (Id_p, PK, E')}{DE; E \vdash^{puse} Id_p \therefore E, Id_p : \textbf{Pkg}\ Id_p, Id_p \overset{pe}{\mapsto} (PK, E')}$$

($PUses\ Alias$)

$$\frac{Id_p \in \operatorname{dom} DE \qquad Pkg = DE\ Id_p \qquad \neg\ (\{Id_p, Id_a\} \subseteq \operatorname{dom} E) \qquad DE \vdash^{pkg} Pkg : (Id_p, PK, E')}{DE; E \vdash^{puse} Id_p\ \textbf{as}\ Id_a \therefore E, Id_p : \textbf{Pkg}\ Id_p, Id_a : \textbf{Pkg}\ Id_p, Id_p \overset{pe}{\mapsto} (PK, E')}$$

The rules for processing package uses (Table 3.18) take a sequence of package references, processing each reference in turn, to yield a new environment. The new environment holds biding 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 3.19 Type rules for package dependencies (*PDep*)

$$\frac{E; Id_p; \overline{PRef} \vdash^{pdep} PDep \therefore E' \qquad E'; Id_p; \overline{PRef} \vdash^{pdep} \overline{PDep} \therefore E''}{E; Id_p; \overline{PRef} \vdash^{pdep} PDep\,\overline{PDep} \therefore E''}\ (PDeps *) \qquad \frac{}{E; Id_p; \overline{PRef} \vdash^{pdep} \epsilon \therefore E}\ (PDeps\ \epsilon)$$

$$\frac{E; Id_p \vdash^{pm} (Id_{p1},\ Id_{p2}, \overline{Id_s}^{1}) \therefore E' \qquad Id_{p1} \neq Id_{p2} \qquad \{Id_{p1}, Id_{p2}\} \subseteq \operatorname{ran} \overline{PRef}}{E; Id_p; \overline{PRef} \vdash^{pdep} Id_{p1}\ \mathbf{merges}\ Id_{p2}\ \mathbf{on}\ \mathbf{(}\overline{Id_s}^{1}\mathbf{)} \therefore E'}\ (PDep\ Merge)$$

$$\frac{E; Id_p \vdash^{po} (Id_{p1},\ Id_{p2}, \overline{Id_s}^{1}) \therefore E' \qquad Id_{p1} \neq Id_{p2} \qquad \{Id_{p1}, Id_{p2}\} \subseteq \operatorname{ran} \overline{PRef}}{E; Id_p; \overline{PRef} \vdash^{pdep} Id_{p1}\ \mathbf{overrides}\ Id_{p2}\ \mathbf{on}\ \mathbf{(}\overline{Id_s}^{1}\mathbf{)} \therefore E'}\ (PDep\ Override)$$

Table 3.19 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 3.20 Type rules for `merge` dependencies

$$\frac{}{E; Id_p \vdash^{pm} (Id_{ps}, Id_{pt}, \epsilon) \therefore E}\ (\text{PMerge } \epsilon)$$

$$\frac{\begin{array}{c} E.PE(Id_{ps}) = (PK_s, E_s) \qquad E.PE(Id_{pt}) = (PK_t, E_t) \\ E_s.SE(Id_s) = (SK, DK, Id_1, VE_1) \qquad E_t.SE(Id_s) = (SK, DK, Id_2, VE_2) \\ VE_r = mergeVEs(VE_1, VE_2) \qquad T_s = \mathbf{Pow\,Set} Id_s \\ SI = (SK, DK, Id_p, VE_r) \qquad E, Id_s : T_s, Id_s \overset{se}{\mapsto} SI \vdash^{pm} (Id_{ps},\ Id_{pt}, \overline{Id_s}) \therefore E' \end{array}}{E; Id_p \vdash^{pm} (Id_{ps},\ Id_{pt}, Id_s\,\overline{Id_s}) \therefore E'}\ (\text{PMerge } *)$$

The type rules for the package merge (Table 3.20) take a source package (Id_{ps}), a target package (Id_{pt}) 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_s 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.3.5). 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 3.21) take a source package (Id_{ps}), a target package (Id_{pt}) 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_s from environments of source and target package (E_s and E_t), which are retrieved from E. The information associated with Id_s in 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_s; the updated environment is then passed to the rule that processes the remaining sequence of overrides.

The rules that process package incorporations (table 3.22) take a sequence of package references to produce a new environment. The non-empty rule (`PIncorporates *`) retrieves the package informa-

Table 3.21 Type rules for `overrides` dependencies

$(POverrides\ \epsilon)$

$$\frac{}{E; Id_p \vdash^{po} (Id_{ps},\ Id_{pt}, \epsilon) \therefore E}$$

$(POverrides\ *)$

$$\frac{\begin{array}{c} E.PE(Id_{ps}) = (PK_s, E_s) \\ E.PE(Id_{pt}) = (PK_t, E_t) \quad E_t \vdash Id_s : T_s \quad T_s = \mathbf{Pow\ Set}\ Id_s \quad E_s \vdash Id_s : T_s \\ E_s.SE(Id_s) = (SK_1, DK_1, Id_{p1}, VE) \quad E_t.SE(Id_s) = (SK_2, DK_2, Id_{p2}, \{\}) \\ SI = (SK_1, DK_1, Id_p, VE) \quad E, Id_s : T_s, Id_s \overset{se}{\mapsto} SI \vdash^{po} (PId_s, PId_t, \overline{Id_s}) \therefore E' \end{array}}{E; Id_p \vdash^{po} (Id_{ps},\ Id_{pt},\ Id_s\, \overline{Id_s}) \therefore E'}$$

Table 3.22 Type rules for package incorporations

$(PIncorporates\ \epsilon)$

$$\frac{}{E; PK \vdash^{pinc} \epsilon \therefore E}$$

$(PIncorporates\ *)$

$$\frac{\begin{array}{c} E.PE(Id_p) = (PK_2, E_p) \quad PK_1 = \mathbf{container} \Rightarrow PK_2 = \mathbf{container} \\ E'' = transferSDEntities(E_p, E) \quad E''; PK \vdash^{pinc} \overline{PRef} \therefore E' \end{array}}{E; PK_1 \vdash^{pinc} Id_p[\mathbf{as}\ Id_a]\ \overline{PRef} \therefore E'}$$

tion 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 structural entities defined in the environment of the incorporated package (E_p) to the current environment E to produce an updated environment E'' using function *transferSDEntities*, (appendix A, section A.3.5). Environment E'' is then passed to the rule that processes the remaining sequence of incorporated packages.

Table 3.23 Type rules for package `extends` declaration

$(PExtends\ \epsilon)$

$$\frac{}{E; Id_p \vdash^{pext} \mathbf{extends(}\epsilon\mathbf{)} \therefore E}$$

$(PExtends\ *)$

$$\frac{\begin{array}{c} E.SE(Id_s) = (SK, SK, Id_o, VE) \\ E \oplus \{Id \overset{se}{\mapsto} (SK, SK, Id_p, VE)\} \vdash^{pext} \mathbf{extends(}\overline{Id,}\mathbf{)} \therefore E' \end{array}}{E; Id_p \vdash^{pext} \mathbf{extends(}Id_s,\ \overline{Id_s,}\mathbf{)} \therefore E'}$$

The rules for processing extensions (table 3.23) take a sequence of extensions, processing each in turn, to yield an updated environment. The empty-sequence rule (`PExtends` ϵ) 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 3.24 Judgements for type system of VCL Structural Diagrams

$E; \overline{AD}; Id_p \vdash^{sd} SD \therefore E'$	Well-formed SD yields E'
$E; \overline{AD}; Id_p \vdash^{sde} \overline{SDE} \therefore E'$	Well-formed sequence of SD elements $\overline{SDE}$ yields environment E'
$E; \overline{AD}; Id_{s\perp} \vdash^{as} \overline{A} \therefore VE$	Sequence of assertions $\overline{A}$ yields variable environment VE
$E; \overline{AD} \vdash^{gcn} GC \therefore E'$	Well-formed global constant GC yields environment E'
$E; \overline{AD}; Id_{s\perp} \vdash^{cn} \overline{C} \therefore VE$	Sequence of constants $\overline{C}$ yields variable environment VE
$E; \overline{AD}; Id_p; T \vdash^{pset} PSet \therefore E'$	Primary Set $PSet$ yields environment E'
$E \vdash^{ped} \overline{PED} \therefore VE$	Sequence of edge definitions $\overline{PED}$ yields variable environment VE
$E; M \vdash^{mtd} TD : T$	Designator TD with multiplicity M yields type T
$E; \overline{AD}; Id_p; T \vdash^{hi} HI \therefore E'$	HI ($HasIn$) yields environment E'
$E; T \vdash^{io} \overline{O} \therefore VE$	Sequence of inside objects $\overline{O}$ yields variable environment VE
$E; Id_p; T \vdash^{is} \overline{PSet} \therefore E'$	Sequence of inside primary sets $\overline{PSet}$ yields environment E'

3.5 Structural Diagrams

Table 3.24 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 ($\vdash^{mtd}$), has inside declarations of primary sets ($\vdash^{hi}$), sequence of inside objects ($\vdash^{io}$) and sequences of inside primary sets ($\vdash^{is}$).

Table 3.25 Type rules for structural diagrams and structural diagram elements

($Ok\ SD$)
$$\frac{E; Id_p \vdash^{sde} \overline{SDE} \therefore E' \qquad Acyclic\ E'.SE \qquad E'; \overline{AD}; \perp \vdash^{as} \overline{A} \therefore VE}{E; \overline{AD}; Id_p \vdash^{sd} \overline{SDE}\,\overline{A} \therefore E', VE}$$

($\overline{SDE}\,*$)
$$\frac{E; \overline{AD}; Id_p \vdash^{sde} SDE \therefore E'' \qquad E''; \overline{AD}; Id_p \vdash^{sde} \overline{SDE} \therefore E'}{E; \overline{AD}; Id_p \vdash^{sde} SDE\ \overline{SDE} \therefore E'}$$

($SDE\ \epsilon$)
$$\frac{}{E; \overline{AD}; Id_p \vdash^{sde} \epsilon \therefore E}$$

($SDE\ Gbl\ Const$)
$$\frac{E; \overline{AD} \vdash^{gcn} GC \therefore E'}{E; \overline{AD}; Id_p \vdash^{sde} GC \therefore E'}$$

($SDE\ RelEdge$)
$$\frac{E \vdash TD_1 : T_1 \qquad E \vdash TD_2 : T_2 \qquad M_1 \neq \mathbf{seq} \qquad M_2 \neq \mathbf{seq}}{E; \overline{AD}; Id_p \vdash^{sde} \mathbf{relEdge}\, Id_{Re}\, (M_1\ TD_1,\ M_2\ TD_2) \therefore E, \{Id_{Re} : \mathbf{Pow\,Pair}\,(T_1, T_2)\}}$$

($SDE\ PSet$)
$$\frac{E; \overline{AD}; Id_p; \mathbf{Obj} \vdash^{pset} PSet \therefore E_s}{E; \overline{AD}; Id_p \vdash^{sde} PSet \therefore E_s}$$

(SDE RSet Uses)
$$\frac{E \vdash Id_p :: Id_s : \mathbf{Pow\,Set}\, Id_s}{E; \overline{AD}; Id_p \vdash^{sde} \uparrow \mathbf{set} Id_p :: Id_s \therefore E}$$

(SDE RSet Inc)
$$\frac{E \vdash Id_s : \mathbf{Pow\,Set}\, Id_s \quad E.SE(Id_s) = (SK, DK, Id_{p2}, VE) \quad Id_{p1} \neq Id_{p2}}{E; \overline{AD}; Id_{p1} \vdash^{sde} \uparrow \mathbf{set} Id_s \therefore E}$$

($SDE\ Derived$)
$$\frac{E \vdash^{sdef} SetDef : T}{E \vdash^{sde} Id_s \leftrightarrow SetDef \therefore E, \{Id_s : T\}}$$

($SDE\ PkgE$)
$$\frac{E \vdash^{ped} \overline{PED} \therefore VE}{E \vdash^{sde} \mathbf{pkg}\, Id\ \mathbf{\{}\overline{PED}\mathbf{\}} \therefore E, VE}$$

Table 3.26 Type rules for constants

$$\frac{E; \overline{AD}; \bot \vdash^{cn} C \therefore VE}{E; \overline{AD} \vdash^{gcn} C \therefore E, VE}\;(Gbl\ Const) \qquad \frac{E \vdash Id_{Cn} : T \qquad E; \overline{AD}; \bot; Id_{Cn} \vdash^{aok} A \therefore VE_a}{E; \overline{AD} \vdash^{gcn} \uparrow \mathbf{const}\, Id_{Cn} \leftrightarrow A \therefore E, VE_a}\;(Gbl\ CRef\ GBl)$$

$$(Gbl\ CRef\ Local)\;\frac{\begin{array}{c} E \vdash Id_s : \mathbf{Pow\,Set}\, Id_s \\ E.SE(Id_s) = (SK, DK, Id_p, VE_s) \qquad E_\varnothing, VE_s \vdash Id_{Cn} : T \qquad E; \overline{AD}; Id_s; Id_{Cn} \vdash^{aok} A \therefore VE_a \end{array}}{E; \overline{AD} \vdash^{gcn} \uparrow \mathbf{const}\, Id_s.Id_{Cn} \leftrightarrow A \therefore E, VE_a}$$

$$(Decl\ Const)\;\frac{E \vdash TD : T}{E; \overline{AD}; Id_{s\bot} \vdash^{cn} \mathbf{const}\, Id_{Cn}\ : TD \therefore \{Id_{Cn} : T\}} \qquad (Decl\ Const\ WConstraint)\;\frac{E \vdash TD : T \qquad E, Id_{Cn} : T; \overline{AD}; Id_{s\bot}; Id_{Cn} \vdash^{aok} A \therefore VE_a}{E; \overline{AD}; Id_{s\bot} \vdash^{cn} \mathbf{const}\, Id_{Cn}\ : TD \leftrightarrow A \therefore \{Id_{Cn} : T\}, VE_a}$$

$$(Cnts\ epsilon)\;\frac{}{E; \overline{AD}; Id_s \vdash^{cn} \epsilon \therefore VE_\varnothing} \qquad (Cnts\ *)\;\frac{E; \overline{AD}; Id_s \vdash^{cn} C \therefore VE_c \qquad E; \overline{AD}; Id_s \vdash^{cn} \overline{C} \therefore VE}{E; \overline{AD}; Id_s \vdash^{cn} C\,\overline{C} \therefore VE_c,\ VE}$$

Table 3.27 Type rules for primary sets

$$(Primary\ Set\ Def)\;\frac{\begin{array}{c} E; \overline{AD}; Id_s \vdash^{cn} \overline{C} \therefore VE_c \\ E \vdash^{ped} \overline{PED} \therefore VE_{pe} \qquad E; \overline{AD}; Id_s \vdash^{as} \overline{A} \therefore VE_a \qquad VE_i = getVE(E, T_i) \\ T_s = \mathbf{Set}\, Id_s \qquad DK = getDK([\bigcirc]) \qquad SI = (SK, DK, Id_p, (VE_c, VE_{pe}, VE_a, VE_i)) \\ E,\ Id_s : \mathbf{Pow}\ T_s, Id_s \overset{se}{\mapsto} SI; \overline{AD}; T_s \vdash^{hi} [\,\mathbf{hasIn}\ \{\overline{(O \mid PSet)}\}\,] \therefore E') \end{array}}{E; \overline{AD}; Id_p; T_i \vdash^{pset} \mathbf{set}\, Id_s\ SK\,[\bigcirc]\,\{\overline{C}\,\overline{PED}\,\overline{A}\}\,[\,\mathbf{hasIn}\ \{\overline{(O \mid PSet)}\}\,] \therefore (E',\ T_s <: T_i)}$$

$$(Set\ Extension)\;\frac{\begin{array}{c} T_s = \mathbf{Set}\, Id_s \\ E \vdash Id_s : \mathbf{Pow}\ T_s \qquad E.SE(Id_s) = (SK, DK, Id_p, VE_s) \qquad getGType(superTy(E, T_s), T_i) = T_{if} \\ E; \overline{AD}; Id_s \vdash^{cn} \overline{C} \therefore VE_c \qquad E \vdash^{ped} \overline{PED} \therefore VE_{pe} \qquad E; \overline{AD}; Id_s \vdash^{as} \overline{A} \therefore VE_a \\ VE_i = getVE(E, T_i) \qquad SI = (SK, DK, Id_p, (VE_s, VE_c, VE_{pe}, VE_a, VE_i)) \\ E \oplus \{Id_s \overset{se}{\mapsto} SI\}; \overline{AD}; T_s \vdash^{hi} [\,\mathbf{hasIn}\ \{\overline{(O \mid PSet)}\}\,] \therefore E' \end{array}}{E; \overline{AD}; Id_p; T_i \vdash^{pset} \downarrow \mathbf{set}\, Id_s\,\{\overline{C}\,\overline{PED}\,\overline{A}\}\,[\,\mathbf{hasIn}\ \{\overline{(O \mid PSet)}\}\,] \therefore (E \oplus \{T_s <: T_{if}\}, E_{hi})}$$

Table 3.28 Type rules for property edge definitions

$$(\overline{PED}\,\epsilon)\;\frac{}{E \vdash^{ped} \epsilon \therefore VE_\varnothing} \qquad (\overline{PED}\,*)\;\frac{E; M \vdash^{mtd} TD : T \qquad E \vdash^{ped} \overline{PED} \therefore VE_2}{E \vdash^{ped} M\ Id_{Pe} \to TD\ \overline{PED} \therefore \{Id_{Pe} : T\}, VE_2}$$

$$(TDOpt)\;\frac{E \vdash^{td} TD \therefore T}{E; \mathbf{opt} \vdash^{mtd} TD : (\mathbf{Opt}\ T)} \qquad (TDOne)\;\frac{E \vdash^{td} TD \therefore T}{E; \mathbf{one} \vdash^{mtd} TD : (T)} \qquad (TDSome)\;\frac{E \vdash^{td} TD \therefore T}{E; \mathbf{some} \vdash^{mtd} TD : (\mathbf{Pow}\ T)}$$

$$(TDMany)\;\frac{E \vdash^{td} TD \therefore T}{E; \mathbf{many} \vdash^{mtd} TD : (\mathbf{Pow}\ T)} \qquad (TDRange)\;\frac{E \vdash^{td} TD \therefore T}{E; Num\,..\,(Num|\mathbf{*}) \vdash^{mtd} TD : (\mathbf{Pow}\ T)} \qquad (TDSeq)\;\frac{E \vdash^{td} TD \therefore T}{E; \mathbf{opt} \vdash^{mtd} TD : (\mathbf{Seq}\ T)}$$

Table 3.29 Type rules for sequences of invariants

$$(\overline{A}\,\epsilon)\quad \frac{}{E;\overline{AD};Id_{s\bot}\vdash^{as}\epsilon\therefore\ VE_{\varnothing}}$$

$$(\overline{A}\,*)\quad \frac{E;\overline{AD};Id_{s\bot};\bot\vdash^{aok}A\therefore\ VE_a \qquad E;\overline{AD};Id_{s\bot}\vdash^{as}\overline{A}\therefore\ VE}{E;\overline{AD};Id_{s\bot}\vdash^{as}A\,\overline{A}\therefore\ VE_a,\ VE}$$

Table 3.30 Type rules for has inside declarations

$$(\mathit{HasInside}\,\epsilon)\quad \frac{}{E;Id_p;T\vdash^{hi}\epsilon\therefore E}$$

$$(\mathit{HasInside}\,*)\quad \frac{E;T\vdash^{io}\overline{O}\therefore\ VE \qquad E;Id_p;T\vdash^{is}\overline{PSet}\therefore E'}{E;Id_p;T\vdash^{hi}\mathbf{hasIn}\,\{\overline{O}\ \overline{PSet}\}\therefore\ E',\,VE}$$

$$(\mathit{HasInObjs}\,\epsilon)\quad \frac{}{E;Id_p;T\vdash^{io}\epsilon\therefore\{\}}$$

$$(\mathit{HasInObjs}\,*)\quad \frac{E;T\vdash^{io}\overline{O}\therefore\ VE}{E;Id_p;T\vdash^{io}\mathbf{object}\ Id_o\ \overline{O}\therefore\ VE,\{Id_o:T\}}$$

$$(\mathit{HasInPSets}\,\epsilon)\quad \frac{}{E;Id_p;T\vdash^{is}\epsilon\therefore E}$$

$$(\mathit{HasInPSets}\,*)\quad \frac{E;Id_p;T\vdash^{pset}PSet\therefore E' \qquad E';Id_p;T\vdash^{is}\overline{PSet}\therefore E''}{E;Id_p;T\vdash^{is}PSet\,\overline{PSet}\therefore E''}$$

Table 3.31 Judgements for typing of assertion diagrams

$E \vdash^{ad} AD \therefore I : T$	AD yields binding $I : T$ in E
$E \vdash^{d} \overline{D} \therefore (VE_v; VE_h)$	Sequence of declarations $\overline{D}$ yields binding sets (VE_v, VE_h)
$E \vdash^{df} DF \therefore (VE_v; VE_h)$	Declarations formula atom DF yields binding sets (VE_v, VE_h)
$E \vdash^{f} \overline{F}$	Sequence of formulas $\overline{F}$ is well-formed in E
$E \vdash^{afs} AFS : T$	Arrows formula source AFS yields type T

Table 3.32 Type rules for assertion diagrams

$(AD\ GBL)$

$$\frac{E \vdash^{d} \overline{D} \therefore (VE_v; VE_h) \qquad E \oplus (VE_v, VE_h) \vdash^{f} \overline{F}}{E \vdash^{ad} \mathbf{AD}\ Id_A\ \mathbf{decls}\ \{\overline{D}\}\ \mathbf{pred}\ \{\overline{F}\} \therefore Id_A : \mathbf{Assertion}[VE_v, VE_h]}$$

$(AD\ LOCAL)$

$$\frac{\begin{array}{c} E \vdash Id_s : \mathbf{Pow\,Set}\ Id_s \qquad E.SE(Id_s) = (SK,\ DK,\ Id_p,\ VE_s) \\ E \oplus VE_s \vdash^{d} \overline{D} \therefore (VE_v; VE_h) \qquad E \oplus (VE_s, VE_v, VE_h) \vdash^{f} \overline{F} \end{array}}{E \vdash^{ad} \mathbf{AD}\ Id_A : Id_s\ \mathbf{decls}\ \{\overline{D}\}\ \mathbf{pred}\ \{\overline{F}\} \therefore Id_A : \mathbf{Assertion}\,[VE_v,\ VE_h]}$$

3.6 Assertion Diagrams

The judgements for ADs are listed in Table 3.31. In the judgements's 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 $\vdash^{d}$, which says that a sequence of declarations ($\overline{D}$) is well-formed and $\vdash^{df}$, which says that a particular declaration formula (DF) is well-formed. The predicate compartment includes judgements for formulas ($\vdash^{f}$) and arrows formula source ($\vdash^{afs}$).

The typing rules for ADs (table 3.32) 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 3.33) build the visible and hidden variable environments. They are follows:

- Rules $\overline{D}\,\epsilon$ and $\overline{D}\,*$ handle a sequence of declaration inductively. Rule $\overline{D}\,\epsilon$ yields the empty variable environments ($\{\}$) for both visible and hidden: there are no declarations to process. Rule $\overline{D}\,*$ retrieves the variable environments from the current declaration ($VE_v,\ VE_h$) and from the remaining declarations ($VE_{vs},\ VE_{hs}$); the variables environments to be yielded by the rule are then merged (operator $\bowtie$), 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_{vf}) are removed in the hidden list (operator $\boxslash$).
- Rules `D Obj` and `D 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 (TD) and then yield a visible binding made of the variable's identifier and appropriate type. Rule `D Obj` considers whether there is an optional qualifier ; type to yield is optional if there is a qualifier (**Opt**T) or the type derived from the type designator otherwise (T). Rule `D Set` also considers whether there is a sequence qualifier; type to yield is sequence of there is a qualifier ($\mathbf{Seq}\,T$) or a powerset otherwise ($\mathbf{Pow}\,T$).

Table 3.33 Type rules for declarations

$$(\overline{D}\,\epsilon)\quad \frac{}{E \vdash^{d} \epsilon \therefore (\{\};\{\})}$$

$$(\overline{D}\,*)\quad \frac{E \vdash^{d} D \therefore (VE_v; VE_h) \qquad E, VE_v, VE_h \vdash^{d} \overline{D} \therefore (VE_{vs}; VE_{hs}) \qquad VE_{vf} = VE_v \bowtie VE_{vs} \qquad VE_{hf} = (VE_h \bowtie VE_{hs}) \boxslash VE_{vf}}{E \vdash^{d} D\,\overline{D} \therefore (VE_{vf}; VE_{hf})}$$

$$(\text{D Obj})\quad \frac{E \vdash^{td} TD : T \qquad (Q = \mathbf{?} \wedge T_f = \mathbf{Opt}\,T \vee Q = \epsilon \wedge T_f = T)}{E \vdash^{d} \mathbf{object}\; Q\; Id_O{:}TD \therefore (\{Id_O : T_f\};\{\})}$$

$$(\text{D Set})\quad \frac{E \vdash^{td} TD : T \qquad Q = [] \wedge T_f = \mathbf{Seq}\,T \vee Q = \epsilon \wedge T_f = \mathbf{Pow}\,T}{E \vdash^{d} \mathbf{set}\; Q\; Id_s{:}TD \therefore (\{Id_s : T_f\};\{\})}$$

$$(D\;DF)\quad \frac{E \vdash^{df} DF \therefore (VE_v; VE_h)}{E \vdash^{d} DF \therefore (VE_v; VE_h)}$$

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

Table 3.34 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_v and VE_h). From the assertion's visible bindings (VE_v), the rule then builds the visible and hidden bindings for the declaration using function *conVEs*, which takes into account the presence of symbol ↑, and from these constructed bindings the rule makes the required substitutions according to what is defined in the sequence of renamings ($\overline{R}$) using function *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 obtains the visible and hidden bindings from the two declarations formulas being combined and then merges them using the operator *mergeves*.

Table 3.34 Type rules for declaration formulas

(DFA Assertion)

$$\frac{\begin{array}{c} E \vdash^{ta} A \therefore Id_A : \mathbf{Assertion}[VE_v; VE_h] \\ (VE_{cv}, VE_{ch}) = consVEs(VE_v, [\uparrow]) \\ (VE_{fv}, VE_{fh}) = applySubs(VE_{cv}, VE_{ch}, [\overline{R},]) \end{array}}{E \vdash^{df} [\uparrow] A [\overline{R},] \therefore (VE_{fv}; VE_{fh})}$$

(DFA OCall)

$$\frac{\begin{array}{c} E \vdash Id_O : T_s \\ Id_s = getSIdFrScalarOrCollection(T_s) \\ E \vdash Id_s\mathbf{.}Id_A : \mathbf{Assertion}[VE_v, VE_h] \\ (VE_{cv}, VE_{ch}) = consVEs(VE_v, [\uparrow]) \\ (VE_{fv}, VE_{fh}) = applySubs(VE_{cv}, VE_{ch}, [\overline{R},]) \end{array}}{E \vdash^{df} [\uparrow] \mathbf{assertion}\ Id_O\mathbf{.}Id_A [\overline{R},] \therefore (VE_{fv}; VE_{fh})}$$

(DFA ClCall)

$$\frac{\begin{array}{c} E \vdash Id_s\mathbf{.}Id_A : \mathbf{Assertion}[VE_v, VE_h] \\ (VE_{cv}, VE_{ch}) = consVEs(VE_v, [\uparrow]) \\ (VE_{fv}, VE_{fh}) = applySubs(VE_{cv}, VE_{ch}, [\overline{R},]) \end{array}}{E \vdash^{df} [\uparrow] \mathbf{assertion}\ Id_s \rightarrow Id_A [\overline{R},] \therefore (VE_{fv}; VE_{fh})}$$

(DFA PkgCall)

$$\frac{\begin{array}{c} E \vdash Id_p\mathbf{::}Id_A : \mathbf{Assertion}[VE_v, VE_h] \\ (VE_{cv}, VE_{ch}) = consVEs(VE_v, [\uparrow]) \\ (VE_{fv}, VE_{fh}) = applySubs(VE_{cv}, VE_{ch}, [\overline{R},]) \end{array}}{E \vdash^{df} [\uparrow] \mathbf{assertion}\ Id_p\mathbf{::}Id_A [\overline{R},] \therefore (VE_{fv}; VE_{fh})}$$

(*DF Neg*)

$$\frac{E \vdash^{df} DF \therefore (VE_v; VE_h)}{E \vdash^{df} \mathbf{(}\neg\ DF\mathbf{)} \therefore (VE_v; VE_h)}$$

(DF Bin)

$$\frac{\begin{array}{c} E \vdash^{df} DF_1 \therefore (VE_{v1}; VE_{h1}) \\ E \vdash^{df} DF_1 \therefore (VE_{v2}; VE_{h2}) \end{array}}{E \vdash^{df} \mathbf{(}DF_1\ FOp\ DF_2\mathbf{)} \therefore (VE_{v1} \bowtie VE_{v2}; VE_{h1} \bowtie VE_{h2})}$$

Table 3.35 Type rules for Formulas (F)

($\overline{F}\ \epsilon$)

$$\frac{}{E \vdash^{f} \epsilon}$$

($\overline{F}\ *$)

$$\frac{\begin{array}{c} E \vdash^{f} F \\ E \vdash^{f} \overline{F} \end{array}}{E \vdash^{f} F\ \overline{F}}$$

(*F Not*)

$$\frac{E \vdash^{f} F}{E \vdash^{f} \neg\ [F]}$$

(F Bin)

$$\frac{\begin{array}{c} E \vdash^{f} F_1 \\ E \vdash^{f} F_2 \qquad FOp \neq \boxdot \end{array}}{E \vdash^{f} [F_1]\ FOp\ [F_2]}$$

(F AF)

$$\frac{\begin{array}{c} E \vdash^{afs} AFS : T \\ E; T \vdash^{peps} \overline{PEP}^1 \end{array}}{E \vdash^{f} AFS\ \mathbf{\{}\overline{PEP}^1\mathbf{\}}}$$

(*F SF Shaded*)

$$\frac{E \vdash^{sdef} SDef : T}{E \vdash^{f} \mathbf{shaded}\ SDef}$$

(F SF Id)

$$\frac{\begin{array}{c} E \vdash Id_s : T_1 \\ E \vdash^{sdef} SDef : T_2 \\ (E \vdash T_2 <: T_1 \\ \vee\ E \vdash T_2 <: T_1) \end{array}}{E \vdash^{f} [\mathbf{shaded}]\ Id_s\ SDef}$$

(F SF Inside)

$$\frac{\begin{array}{c} E \vdash^{td} TD : T_1 \\ E \vdash^{sexp} SExp : T_2 \\ E \vdash T_1 <: T_2 \end{array}}{E \vdash^{f} \mathbf{set}\ TD\ \mathbf{hasIn}\ \{SExp\}}$$

(*AFS SE*)

$$\frac{E \vdash^{td} TD : \mathbf{Pow}\ T_2}{E \vdash^{f} \mathbf{set\ shaded}\ TD}$$

Table 3.36 Type rules for Arrows Formula Source (production *AFS*)

(*AFS SE*)

$$\frac{E \vdash^{se} SE : T}{E \vdash^{afs} SE : T}$$

(*AFS SetId*)

$$\frac{E \vdash Id_s : T}{E \vdash^{afs} \mathbf{set}\ Id_s : T}$$

(*AFS SDef*)

$$\frac{E \vdash^{sdef} SDef : T}{E \vdash^{afs} SDef : T}$$

(*AFSB Un Card*)

$$\frac{E \vdash^{afs} AFS : \mathbf{Pow}\ T}{E \vdash^{afs} \#\ AFS : \mathbf{Int}}$$

(*AFS Un Dom*)

$$\frac{E \vdash^{afs} AFS : \mathbf{Pow\ Pair}\ (T_1, T_2)}{E \vdash^{afs} \leftarrow\ AFS : \mathbf{Pow}\ T_1}$$

(*AFS Un Ran*)

$$\frac{E \vdash^{afs} AFS : \mathbf{Pow\ Pair}\ (T_1, T_2)}{E \vdash^{afs} \rightarrow\ AFS : \mathbf{Pow}\ T_2}$$

(*AFSB Un The*)

$$\frac{E \vdash^{afs} AFS : \mathbf{Opt}\ T}{E \vdash^{afs} \circledast\ AFS : T}$$

Table 3.37 Judgements for typing of VCL models

$\vdash^{vcl} VCL$	VCL is well-formed VCL model
$DE \vdash^{pkg} Pkg \therefore (Id_p, PK, E)$	Well-formed Pkg yields triple (Id_p, PK, E)

Table 3.38 Types rules for overall VCL models made of packages

(OK VCL)

$$\frac{\begin{array}{c} PkgsOnce\,(VCL) \\ DE = buildDE\,(VCL) \qquad AyclicPkgs(DE) \\ VCL = \overline{Pkg} \qquad DE \vdash^{pkg} Pkg : \overline{(Id_p, PK, E)} \end{array}}{\vdash^{vcl} VCL}$$

(OK Pkg)

$$\frac{\begin{array}{c} DE \vdash^{pd} PD \therefore (Id_p, PK, E) \\ E; \overline{AD}; Id_p \vdash^{sd} SD \therefore E' \end{array}}{DE \vdash^{pkg} PD\ SD\ \overline{AD} \therefore (Id_p, PK, E')}$$

3.7 VCL Models

We now describe the typing rules for overall VCL models . The judgements are listed in table 3.37. 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 3.38 presents the type rules for VCL models. The rules are as follows:

- Rule `OK VCL` checks well-formedness of a VCL model. A VCL model is well-formed provided: (a) all its packages are defined once (predicate *PkgsOnce*, appendix A, sec. A.2.4), (b) that the graph formed by the package uses and incorporates dependencies is acyclic (predicate *AyclicPkgs*, appendix A, sec. A.2.5) in the diagram environment that is built from the VCL model (function *buildDE*, appendix A, sec. A.3.2), (c) and the current package is well-formed in the diagram environment.
- Rule `OK Pkg` specify well-formedness of a VCL Package. This amounts to type-check PD, SD and ADs. The rule for checking PD ($\vdash^{pd}$) yields an environment (E), which is then used to check the the SD and all relevant ADs, resulting in an updated environment.

References

[AGK11] Nuno Amálio, Christian Glodt, and Pierre Kelsen. Building VCL models and automatically generating Z specifications from them. In *FM 2011, tool paper (to appear)*, 2011. available at `http://bit.ly/flbMeH`.

[AK10] Nuno Amálio and Pierre Kelsen. Modular design by contract visually and formally using VCL. In *VL/HCC 2010*, 2010.

[AKM10] Nuno Amálio, Pierre Kelsen, and Qin Ma. Specifying structural properties and their constraints formally, visually and modularly using VCL. In *EMMSAD 2010*, volume 50 of *LNBIP*, pages 261–273. Springer, 2010.

[AKMG10] Nuno Amálio, Pierre Kelsen, Qin Ma, and Christian Glodt. Using VCL as an aspect-oriented approach to requirements modelling. *TAOSD*, VII:151–199, 2010.

[Amá07] Nuno Amálio. *Generative frameworks for rigorous model-driven development*. PhD thesis, Dept. Computer Science, Univ. of York, 2007.

[Amá11] Nuno Amálio. VCL model of the secure simple bank case study. Technical Report TR-LASSY-11-05, Univ. of Luxembourg, 2011. `http://bit.ly/q1LrPj`.

[APS05] Nuno Amálio, Fiona Polack, and Susan Stepney. An object-oriented structuring for Z based on views. In *ZB 2005*, volume 3455 of *LNCS*, pages 262–278. Springer, 2005.

[BKPPT01] Paolo Bottoni, Manuel Koch, Francesco Parisi-Presicce, and Grabriele Taentzer. A visualisation of OCL using collaborations. In *UML 2001*, volume 2185, pages 257–271, 2001.

[Car04] Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*. CRC Press, 2004.

[Cla99] Tony Clark. Typechecking UML static models. In *UML'99*, volume 1723 of *LNCS*. Springer, 1999.

[EW06] Karsten Ehrig and Jessica Winkelmann. Model transformation from visual OCL to OCL using graph transformation. *ENTCS*, 152:23–37, 2006.

[FFH05] Andrew Fish, Jean Flower, and John Howse. The semantics of augmented constraint diagrams. *Journal of Visual Languages and Computing*, 16:541–573, 2005.

[ISO02] ISO. Information technology—Z formal specification notation—syntax, type system and semantics, 2002. ISO/IEC 13568:2002, Int. Standard.

[Jac06] Daniel Jackson. *Software Abstractions: logic, lanaguage, and analysis*. MIT Press, 2006.

[Ken97] Stuart Kent. Constraint diagrams: Visualizing invariants in object-oriented methods. In *Proc. of OOPSLA'97*, pages 327–341. ACM Press, 1997.

[Kya05] Marcel Kyas. An extended type system for ocl supporting templates and transformations. In *FMOODS 2005*, volume 3535 of *LNCS*, pages 83–98. Springer, 2005.

[LP99] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed? *ACM transactions on Programming Languages and Systems*, 21(3):502–526, 1999.

[OMG10] OMG. *UML infrastructure Specification, v2.3*, 2010.

[Sch02] Andy Schürr. A new type checking approach for OCL version 2.0? In *Object Modeling with the OCL*, volume 2263 of *LNCS*, pages 431–434. Springer, 2002.

[Spi92] J. M. Spivey. *The Z notation: A reference manual*. Prentice-Hall, 1992.

[TVSK00] Ian Toyn, Samuel Valentine, Susan Stepney, and Steve King. Typechecking Z. In *ZB 2000*, volume 1878 of *LNCS*, pages 264–285. Springer, 2000.

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:

$$\begin{array}{l} E_1, E_2 = ((VE_1, VE_2), PE_1 \cup PE_2, SE_1 \cup SE_2, SubE_1 \cup SubE_2) \\ \text{where, } E_1 = (VE_1, PE_1, SE_1, SubE_1) \wedge E_2 = (VE_2, PE_2, SE_2, SubE_2) \end{array}$$

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

$$VE_1, VE_2 = VE_1 \cup VE_2 \Leftrightarrow \mathrm{dom}\, VE_1 \cap \mathrm{dom}\, VE_2 = \varnothing$$

Another operation on variable environments is $\bowtie$, 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:

$$\_ \bowtie \_ : VE \times VE \nrightarrow VE$$

This is defined inductively by the following equations:

$$\begin{array}{l} \{\} \bowtie VE = VE \\ (\{id : T\} \cup VE_1) \bowtie VE_2 = VE_1 \bowtie (VE_2 \cup \{Id : T\}) \Leftrightarrow id \notin \mathrm{dom}\, VE_2 \vee VE_2(Id) = T \end{array}$$

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:

$$\_ \boxslash \_ : VE \times VE \nrightarrow VE$$

This is defined by the following equation:

$$VE_1 \boxslash VE_2 = VE_1 \setminus VE_2 \Leftrightarrow (\forall\, Id \in (\mathrm{dom}\, VE_1 \cap \mathrm{dom}\, VE_2) \bullet VE_1(Id) = VE_2(Id))$$

E, VE means that a variable environment is added to an environment. This is defined as:

$$E, VE = \begin{cases} (VE_E \cup VE, PE, SE, SubE) & \text{If } E = (VE_E, PE, SE, SubE) \wedge \neg\, (\mathrm{dom}\, VE \subseteq \mathrm{dom}\, VE_E) \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$E, T_1 <: T_2$ means that a subtyping tuple is added to an environment. This is defined as:

$$E, T_1 <: T_2 = (VE, PE, SE, SubE \cup \{T_1 \mapsto T_2\}) \quad \text{where, } E = (VE, PE, SE, SubE)$$

$E, Id \overset{se}{\mapsto} (SK, DK, Id, VE)$ means that a set environment binding is added to an environment. This is defined as:

$$E, Id \overset{se}{\mapsto} (SK, DK, Id, VE) =$$
$$\begin{cases} (VE, PE, SE \cup \{(SK, DK, Id, VE)\}, SubE) & \text{If } E = (VE, PE, SE, SubE) \wedge Id \notin \mathrm{dom}\, E.SE \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$E, Id \overset{pe}{\mapsto} (PK, E)$ means that a package environment binding is added to an environment. This is defined as:

$$E, Id \overset{pe}{\mapsto} (PK, E) =$$
$$\begin{cases} (VE, PE \cup \{(PK, E)\}, SE, SubE) & \text{If } E = (VE, PE, SE, SubE) \wedge Id \notin \mathrm{dom}\, E.SE \\ \mathit{undefined} & \text{otherwise} \end{cases}$$

$E \oplus VE$ means that an environment is overridden with a set of variable bindings. This is defined as:

$$E \oplus VE_2 = (VE_1 \oplus VE_2, PE, SE, SubE) \quad \text{where, } E = (VE_1, PE, SE, SubE)$$

A.2 Predicates

A.2.1 Predicate *Ayclic*

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

$$\mathit{Acyclic}\,(R) \Leftrightarrow R \in \{rel : X \leftrightarrow X \mid rel^{+} \cap \mathrm{id}\, X = \varnothing\}$$

A.2.2 Predicates *IsSeqOfPEP* and *IsSeqOfPEM*

$$\mathit{IsSeqOfPEP}(PE\, \overline{PEP}) \Leftrightarrow PE = PEP \wedge (\overline{PE} = \epsilon \vee \mathit{IsSeqOfPEP}(\overline{PE}))$$
$$\mathit{IsSeqOfPEM}(PE\, \overline{PEP}) \Leftrightarrow PE = PEM \wedge (\overline{PE} = \epsilon \vee \mathit{IsSeqOfPEM}(\overline{PE}))$$

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).

$$\mathbf{NoClashes} : \mathbb{P}(\overline{PRef})$$
$$\mathbf{NoClashes} : \mathbb{P}(\overline{PRef}, \overbrace{Id})$$
$$\mathit{NoClashes}\,(\overline{PRef}) \Leftrightarrow \mathit{NoClashes}_0\,(\overline{PRef}, \varnothing)$$
$$\mathit{NoClashes}_0\,(\epsilon, S)$$
$$\mathit{NoClashes}_0\,(Id_p\,[\mathbf{as}\, Id_a]\,\overline{PRef}, S) \Leftrightarrow Id_p \notin S \wedge \mathit{NoClashes}_0\,(\overline{PRef}, S \cup \{Id_p\})$$

A.2.4 Predicate *PkgsOnce*

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

$$
\begin{array}{l}
\textbf{PkgsOnce} : \mathbb{P}(\overline{Pkg}) \\
\textbf{PkgsOnce}_\mathbf{0} : \mathbb{P}(\overline{PRef}, \overbrace{Id}) \\
\quad PkgsOnce\ (\overline{Pkg}) \Leftrightarrow PkgsOnce_0\ (\overline{Pkg}, \{\}) \\
\quad PkgsOnce_0\ (Pkg\ \overline{Pkg}, S) \Leftrightarrow \overline{Pkg} = \epsilon\ \vee \\
\qquad (\overline{Pkg} = (PD\ SD\ \overline{AD})\ \overline{Pkg_2} \wedge Id_p = getIdOfPD(PD) \wedge Id_p \notin S \wedge PkgsOnce_0\ (\overline{Pkg_2}, S \cup \{Id_p\}))
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{BuildPkgDepG} : DE \to (Id \leftrightarrow Id) \\
\quad BuildPkgDepG(\{\}) = \{\} \\
\quad BuildPkgDepG(\{Id_p \mapsto Pkg\} \cup DE) = R \Leftrightarrow Pkg = PD\ SD\ \overline{AD} \wedge I = getIdsOfIncs(PD) \\
\qquad \wedge\ U = getIdsOfUses(PD) \wedge R = \{Id_p\} \times I \cup \{Id_p\} \times U \cup BuildPkgDepG(DE) \\
\textbf{AcyclicPkgs} : \mathbb{P}(DE) \\
\quad AcyclicPkgs(DE) \Leftrightarrow Acyclic(BuildPkgDepG\ (DE))
\end{array}
$$

A.3 Auxiliary Functions

A.3.1 Functions to extract information from PDs

$$
\begin{array}{l}
\textbf{getIdOfPD} : PD \to Id \\
\quad getIdOfPD(PK\ \textbf{package}\ Id\ [\textbf{uses}\ \overline{PRef,}]\ [\textbf{incorporates}\overline{PRef,}]\boldsymbol{\{}\overline{PDep}\ [PExts]\boldsymbol{\}}) = Id \\
\textbf{getIdsOfIncs} : PD \to \overbrace{Id} \\
\quad getIdsOfIncs(PK\ \textbf{package}\ Id\ [\textbf{uses}\ \overline{PRef_1,}]\ [\textbf{incorporates}\overline{PRef_2,}]\boldsymbol{\{}\overline{PDep}\ [PExts]\boldsymbol{\}}) = getIdsOfPRefSeq(\overline{PRef_2,}) \\
\textbf{getIdsOfUses} : PD \to \overbrace{Id} \\
\quad getIdsOfUses(PK\ \textbf{package}\ Id\ [\textbf{uses}\ \overline{PRef_1,}]\ [\textbf{incorporates}\overline{PRef_2,}]\boldsymbol{\{}\overline{PDep}\ [PExts]\boldsymbol{\}}) = getIdsOfPRefSeq(\overline{PRef_1,}) \\
\textbf{getIdsOfPRefSeq} : \overline{PRef,} \to \overbrace{Id} \\
\quad getIdsOfPRefSeq(\epsilon) = \{\} \\
\quad getIdsOfPRefSeq(Id_p\ [\textbf{as}\ Id_a]\ \overline{PRef,}) = \{Id_p\} \cup getIdsOfPRefSeq(\overline{PRef,})
\end{array}
$$

A.3.2 Function *buildDE*

$$
\begin{array}{l}
\textbf{buildDE} : \overline{Pkg} \nrightarrow DE \\
\quad buildDE(\epsilon) = \{\} \\
\quad buildDE(Pkg\ \overline{Pkg}) = \{Id_p \mapsto Pkg\} \cup buildDE(\overline{Pkg}) \Leftrightarrow Pkg = PD\ SD\ \overline{AD} \wedge Id_p = getIdOfPD(PD)
\end{array}
$$

A.3.3 Function *getGType*

The function *getGType* gets the greatest type between two types ordered by the subtyping relation:

$$
\begin{array}{l}
\mathbf{getGType} : E \times \mathit{Type} \times \mathit{Type} \nrightarrow \mathit{Type} \\
\quad \mathit{getGType}\,(E, T_1,\ T_2)\ = \begin{cases} T_1 & \text{If } E \vdash T_2\ <:\ T1 \\ T_2 & \text{If } E \vdash T_1\ <:\ T2 \\ \mathit{undefined} & \text{otherwise} \end{cases}
\end{array}
$$

A.3.4 Function *superTy*

The function *superTy* gets the super type of some type:

$$
\begin{array}{l}
\mathbf{superTy} : E \times \mathit{Type} \nrightarrow \mathit{Type} \\
\quad \mathit{superTy}(E, T_1) = T_2 \Leftrightarrow (T_1, T_2) \in E.\mathit{SubE}
\end{array}
$$

A.3.5 Functions producing variable environments (VEs)

The function *getVE* extracts variable environments from set types:

$$
\begin{array}{l}
\mathbf{getVE} : T \times E \to VE \\
\quad \mathit{getVE}\,(T, E)\ = \begin{cases} VE & \text{If } T\ =\ \mathbf{Set}\,\mathit{Id}_s\ \wedge\ E.SE(\mathit{Id}_s) = (SK,\ DK,\ \mathit{Id}_p,\ VE) \\ \{\} & \text{otherwise} \end{cases}
\end{array}
$$

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:

$$
\begin{array}{l}
\mathbf{consVEs}\ :\ VE \times \uparrow_{\perp}\ \to VE\ \times\ VE \\
\quad \mathit{consVEs}\,(VE, \uparrow_{\perp}) = \begin{cases} (VE, \{\}) & \text{if } \uparrow_{\perp} = \uparrow) \\ (\{\}, VE) & \text{if } \uparrow_{\perp} = \perp) \end{cases}
\end{array}
$$

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:

$$
\begin{array}{l}
\mathbf{mergeVEs} : VE \times VE \nrightarrow VE \\
\quad \mathit{mergeVEs}\,(VE_1,\ VE_2)\ =\ VE_1 \cup VE_2 \\
\quad \Leftrightarrow \forall\, \mathit{id} : \mathit{Id} \mid \mathit{id} \in (\operatorname{dom} VE_1 \cap \operatorname{dom} VE_2) \bullet\ VE_1\, \mathit{id}\ =\ VE_2\, \mathit{id}
\end{array}
$$

A.3.6 Function *transferSDEntities*

The function *transferSDEntities* transfer entities defined in a structural diagram (SD)from one environment to the other.

The auxiliary predicate *IsNewSDEntityOf* tells whether some type corresponds to a SD definition that is not defined in some environment.

$$
\begin{array}{l}
\mathbf{IsNewSDEntityOf}\,\_ : \mathbb{P}\,(\mathit{Id} \times T \times E) \\
\quad \mathit{IsNewSDEntityOf}(\mathit{Id}, T, E) \Leftrightarrow (T = \mathbf{Pow}\, T_2 \vee T = \mathbf{Set}\, \mathit{Id}_s) \wedge \neg\, \mathit{Id} \in E.VE
\end{array}
$$

This says that a type corresponds to a SD entity if it is either a powerset or set type. In addition, it says that the identifier must not be defined in the environment.

$$
\begin{array}{l}
\textbf{transferSDEntities} : E \times E \nrightarrow E \\
\quad \mathit{transferSDEntities}\,(\{\}, E) = E \\
\quad \mathit{transferSDEntities}\,(\{id \mapsto T\} \cup E, E') = \{id \mapsto T\}, \mathit{transferSDEntities}\,(E, E') \\
\qquad \Leftrightarrow \mathit{IsNewSDEntityOf}\,(Id, T, E') \\
\quad \mathit{transferSDEntities}\,(\{id \mapsto T\} \cup E, E') = \mathit{transferSDEntities}\,(E, E') \\
\qquad \Leftrightarrow \neg\ \mathit{IsNewSDEntityOf}\,(Id, T, E')
\end{array}
$$

A.3.7 Function *getDK*

The function *getDK* extracts the definitional kind:

$$
\begin{array}{l}
\textbf{getDK} : [\bigcirc] \rightarrow DK \\
\quad \mathit{getDK}\ (\bigcirc)\ = \textbf{def} \\
\quad \mathit{getDK}\ (\epsilon)\ = \textbf{notDef}
\end{array}
$$

A.3.8 Functions to extract information from ADs

The following functions extract the AD identifier, set identifier and declarations from *ADs*:

$$
\begin{array}{l}
\mathit{getIdOfAD} : AD \rightarrow Id \\
\quad \mathit{getIdOfAD}(\textbf{AD}\ Id_A\ [: Id_s]\ \textbf{decls}\ \{\overline{D}\}\ \textbf{pred}\ \{\overline{F}\}) = Id_A \\
\mathit{getSIdOfAD} : AD \nrightarrow Id_\perp \\
\quad \mathit{getSIdOfAD}(\textbf{AD}\ Id_A : Id_s\ \textbf{decls}\ \{\overline{D}\}\ \textbf{pred}\ \{\overline{F}\}) = Id_s \\
\quad \mathit{getSIdOfAD}(\textbf{AD}\ Id_A\ \textbf{decls}\ \{\overline{D}\}\ \textbf{pred}\ \{\overline{F}\}) = \perp \\
\mathit{getDeclsOfAD} : AD \rightarrow \overline{D} \\
\quad \mathit{getDeclsOfAD}(\textbf{AD}\ Id_A\ [: Id_s]\ \textbf{decls}\ \{\overline{D}\}\ \textbf{pred}\ \{\overline{F}\}) = \overline{D}
\end{array}
$$

The following functions get the set of ADs that are included in some *AD*:

$getDepsOfAD : AD \times \overline{AD} \times Id_{\perp} \rightarrow \widehat{AD}$
$getDepsOfAD\ (AD, \overline{AD}, Id_{s\perp}) = getADsOfDecls\ (getDeclsOfAD\ (AD), \overline{AD}, Id_{s\perp})$

$getADsOfDecls : \overline{D} \times \overline{AD} \times Id_{\perp} \rightarrow \widehat{AD}$
$getADsOfDecls\ (\epsilon, \overline{AD}, Id_{s\perp}) = \{\}$
$getADsOfDecls\ (D\ \overline{D}, \overline{AD}, Id_{s\perp}) =$
$getADsOfDecl(D, \overline{AD}, Id_{s\perp}) \cup getADsOfDecls\ (\overline{D}, \overline{AD}, Id_{s\perp})$

$getADsOfDecl : Decl \times \overline{AD} \times Id_{\perp} \rightarrow \widehat{AD}$
$getADsOfDecl\ (DV\ Id\ \mathbf{:}\ TD, \overline{AD}, Id_{s\perp}) = \{\}$
$getADsOfDecl\ (DF, \overline{AD}, Id_{s\perp}) = getADsOfDF\ (DF, \overline{AD}, Id_{s\perp})$

$getADsOfDF : DF \times \overline{AD} \times Id_{\perp} \rightarrow \widehat{AD}$
$getADsOfDF\ ([\uparrow]\mathbf{assertion}\ Id_A\ [\overline{R},], \overline{AD}, Id_{s\perp}) = \{findAD(\overline{AD}, Id_{s\perp}, Id_A)\}$
$getADsOfDF\ ([\uparrow]\mathbf{assertion}\ Id_o\mathbf{.}Id_A\ [\overline{R},], \overline{AD}, Id_{s\perp}) = findLADsWithName(\overline{AD}, Id_A)$
$getADsOfDF\ ([\uparrow]\mathbf{assertion}\ Id_s \rightarrow Id_A\ [\overline{R},], \overline{AD}, Id_{s\perp}) = \{findAD(\overline{AD}, Id_s, Id_A)\}$
$getADsOfDF\ (\neg\ \mathbf{(}DF\mathbf{)}, \overline{AD}, Id_{s\perp}) = getADsOfDF(DF, \overline{AD}, Id_{s\perp})$
$getADsOfDF\ (\mathbf{(}DF_1\ FOp\ DF_2\mathbf{)}, \overline{AD}, Id_{s\perp}) = getADsOfDF(DF_1, \overline{AD}, Id_{s\perp})$
$\cup\ getADsOfDF(DF_2, \overline{AD}, Id_{s\perp})$

$getMatchingAD : AD \times Id \rightarrow \widehat{AD}$

$$getMatchingAD(AD, Id_A) = \begin{cases} \{AD\} & \text{If } getSIdOfAD(AD) \neq \perp \wedge\ getIdOfAD(AD) = Id_A \\ \{\} & \text{otherwise} \end{cases}$$

$findLADsWithName : \overline{AD} \times Id \nrightarrow \widehat{AD}$
$findLADsWithName(\epsilon, Id_A) = \{\}$
$findLADsWithName(AD\ \overline{AD}, Id_A) = getMatchingAD(AD, Id_A) \cup findLADsWithName(\overline{AD}, Id_A)$

A.3.9 Functions for AD lookup

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

$\mathbf{findAD} : \overline{AD} \times Id_{s\perp} \times Id_A \nrightarrow AD$
$findAD\ (\overline{AD}, \perp, Id_A) = findGblAD(\overline{AD}, Id_A)$
$findAD\ (\overline{AD}, Id_s, Id_A) = findLAD(\overline{AD}, Id_s, Id_A)$
$\mathbf{findGblAD} : \overline{AD} \times Id_A \rightarrow AD$
$findGblAD\ (AD, Id_A) = AD \Leftrightarrow getIdOfAD(AD) = Id_A$
$findGblAD\ (AD\ \overline{AD}, Id_A) = AD \Leftrightarrow getIdOfAD(AD) = Id_A$
$findGblAD\ (AD\ \overline{AD}, Id_A) = findGblAD\ (\overline{AD}, Id_A) \Leftrightarrow getIdOfAD(AD) \neq Id_A$
$\mathbf{findLAD} : \overline{AD} \times Id_s \times Id_A \nrightarrow AD$
$findLAD\ (AD, Id_s, Id_A) = AD \Leftrightarrow getSIdOfAD(AD) = Id_s \wedge getIdOfAD(AD) = Id_A$
$findLAD\ (AD\ \overline{AD}, Id_s, Id_A) = AD \Leftrightarrow getSIdOfAD(AD) = Id_S \wedge getIdOfAD(AD) = Id_A$
$findLAD\ (AD\ \overline{AD}, Id_s, Id_A) = findLAD(\overline{AD}, Id_s, Id_A)$
$\Leftrightarrow getIdOfAD(AD) \neq Id_A \vee getSIdOfAD(AD) \neq Id_s$

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:

$\mathbf{getFAId} : Id_{c\perp} \times Id_A \rightarrow Id_A$
$\mathbf{idCnConstraint} : CnId \times AId \rightarrow AId$

$$getFAId\ (Id_{c\perp}, Id_A) = \begin{cases} Id_A & \text{If } Id_{c\perp} = \perp \\ idCnConstraint(CnId_{\perp}, AId) & \text{otherwise} \\ & \end{cases}$$

A.3.10 Functions for substitutions

The following functions deal with substitutions in variable environments:

$$
\begin{array}{l}
\textbf{applySubs} \;:\; VE \times VE \times \mathbb{P}\, Renaming \rightarrow VE \times VE \\
\quad applySubs\,(VE_v, VE_h, Rens) = (doSubs(VE_v, Rens), doSubs(VE_h, Rens)) \\
\textbf{substitute} \;:\; VE \times Renaming \rightarrow VE \\
\textbf{doSubs} \;:\; VE \times \mathbb{P}_1\, Renaming \rightarrow VE \\
\quad substitute\,(VE, idn/ido) = \begin{cases} VE & \text{If } ido \notin VE \vee idn \in VE \\ (VE \setminus \{(ido, VE\, ido)\}) \cup \{(idn, VE\, ido)\} & \text{otherwise} \\ & \end{cases} \\
\quad doSubs\,(VE,) = VE \\
\quad doSubs\,(VE, Renaming \cup Rens) = doSubs(substitute(VE, Renaming), Rens)
\end{array}
$$

A.3.11 Function *getSIdFrScalarOrCollection*

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

$$
\begin{array}{l}
\textbf{getSIdFrScalarOrCollection} \;:\; Type \nrightarrow Id \\
\quad getSIdFrScalarOrCollection\,(\textbf{Set}\, Id_s) = Id_s \\
\quad getSIdFrScalarOrCollection\,(\textbf{Pow}\,\textbf{Set}\, Id_s) = Id_s \\
\quad getSIdFrScalarOrCollection\,(\textbf{Seq}\,\textbf{Set}\, Id_s) = Id_s
\end{array}
$$

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
--==========================================================================

-- Signature of all names
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 signture 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 orverride 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
```

```
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 {}
```

```
--=============================================================================
-- Name: 'SetElement'
--
-- Description:
--    + Defines a set element
--    + Either a single object or a pair
--=============================================================================

abstract sig SetElement {
}


--=============================================================================
-- Name: 'VCLObject'
--
-- Description:
--    + A named VCL object
--    + Elements that can be inside a blob in either primitive or derived blobs
--=============================================================================

sig VCLObject extends SetElement {
   id : Name
}

--=============================================================================
-- Name: 'Pair'
--
-- Description:
--    + Represents a pair made of two named objects
--=============================================================================

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

--============================================================================
-- Name: 'Assertion'
--
-- Description:
--    + Defines assertions whose symbol is the elongated hexagon.
--============================================================================
sig Assertion {
   idAssertion  : Name
}

--=========================================================================
-- Name: 'TypeDesignator'
--
-- Description:
--    + 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 operarator used in edges.
--============================================================================
abstract sig EdgeOperator {
}

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

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

--============================================================================
-- Name: 'EdgeOperatorUnary'
--
-- Description:
--    + Defines edge operarator used in modifer 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 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 folowing 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 definitons!
-- 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 nside.
--=============================================================================
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,
}{
-- The constants must be definition constants
lConstants.kind in ConstDefinition
}

--
-- An object should be in at most one blob (the inverse of the relation is a partial function)
fact objInAtMostOneBlob {
all n : VCLObject | lone n.~hasInsideO
}

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

```
-- 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 obly
-- 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 preeced by the symbol '↑'
--    + Blob references have a reference to a package. The package
--    name is sepearated from the blob's name using '::'
--===========================================================================
sig BlobReference extends PrimaryBlob {
pkgId : Name
}
--===========================================================================
-- Name: 'PkgBlob'
--
-- Description:
--    + Defines a package blob
--    + Package blobs are represented in bold and double-lined.
--===========================================================================
sig PkgBlob extends BlobWithProps {
}

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

```
-- Name: 'PropEdgeDef' (Property Edge Definition)

-- Description:
--    + Defines properties of blobs.
--    + Relates one blob (having property) to another (type of property).
--    + A property edge has a 'Blob' as target.
--    + A property edge may have a multiplicity.
--
--   ----------          0..*-------
--   |PropEdge|------------->|Blob |
--   ----------       target -------
--===========================================================================

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.lConstants+b.lInvariants+b.lProps+(b.hasInsideO))
| nameOf [e1] = nameOf [e2]
       =>  e1 = e2
}

--
-- All global names must be unique
fact GblNamesAreUnique {
   all e1, e2 :
   (Blob+(Assertion-(PrimaryBlob.lInvariants))+
      RelEdge+(Constant-(PrimaryBlob.lConstants)))
 | nameOf[e1] = nameOf[e2] implies e1 = e2
}

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

```
-- 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 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 '[]' preceedes 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 {
```

```
}

--===========================================================================
-- Name: 'DeclRefKind'
--
-- Description:
--    + Defines kind of a declarations reference.
--===========================================================================
abstract sig DeclRefKind {
}

--===========================================================================
-- Name: 'DeclRefKindCall', 'DeclRefKindCallNew', 'DeclRefKindCallPkg'
--
-- Description:
--    + 'DeclRefKindSimple' call to an operation defined in the same scope
--    + 'DeclRefKindCall' call to a local operation of kind 'Update' or 'Delete'
--    + 'DeclRefKindCallNew' call to a local operation of kind 'New'
--    + 'DeclRefKindCallPkg' call to an operation defined in other package
--===========================================================================

one sig
DeclRefKindSimple,
  DeclRefKindCall,
DeclRefKindCallNew,
DeclRefKindCallPkg
extends DeclRefKind {}

--===========================================================================
-- Name: 'RenamingExp'
--
-- Description:
--    + Defines a renaming expression, denoted in logic as [u/y]
--      where expression u denoted as the susbtition for variable y.
--===========================================================================
sig RenamingExp {
subExp : Name, -- Substituting expression
  varToSub : Name -- Variable to substitute
}

--===========================================================================
-- Name: 'DeclFormulaAtom'
--
-- Description:
--    + A declarations formula atom holds references to assertions or contracts
--    + 'refId' is the identifier of referenced assertion or contract
--    + The import is represented by the symbol '↑'
--    + Optional 'callObj' indicates a call a local operation on an object
--       represented as :"a.op".
--    + Optional field 'origin' indicates origin of the operation (blob or package).
--    + Renaming expressions represented as '[t/x,u/y]'. In Ecore,
--    'RenamingExp' is just a String.
--===========================================================================
```

```
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, DFSComp 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'
```

```
--
-- Description:
--    + Defines the source of an arrows formula
--    + It cain either be: obj, blob or pair
--============================================================================
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
--     langauge (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
```

```
}

--=========================================================================
-- Name: 'BlobFormula'
--
-- Description:
--    + Defines a 'Blob' formula.
--=========================================================================
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
}
```