The visual contract language: abstract modelling of software systems visually, formally and modularly

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

TR-LASSY-10-03

Contents

Chapter 1

Introduction

1.1 Background

Thinking, designing and communicating with pictures are recognised essential activities in traditional branches of engineering [Fer77]. Modern day software engineering practice reflects this prominence: informal and ephemeral diagrams are used as discussion sketches; visual languages like UML [SS86, AHGT06] are widely used to document specification and designs at different levels of abstraction.

Visual languages like UML are known as *semi-formal* methods, because they have a formal syntax but no formal semantics. Although there have been successful formalisations of semantics for such languages (e.g subsets of UML, see [Amá07]), they are mostly used without a formal semantics. The lack of formal semantics brings numerous problems [EFLR98]: (a) it is difficult to be precise and have a good understanding of what is being specified, (b) resulting models are prone to ambiguity and inconsistency, and (c) it is not possible to precisely *predict* or *calculate* consequences of modelled artifact as is done in other engineering disciplines using mathematics. Another problem is that visual semi-formal methods cannot express a large number of properties diagrammatically; this is why UML is accompanied by the textual Object Constraint Language (OCL).

Formal methods, on the other hand, strive for soundness, rigour and correctness, providing mathematically rigorous approaches to software engineering. Formal notations have mathematically sound foundations, enabling both precise description, and prediction through calculation. Despite some success stories [CGR95], formal methods have not been embraced by industry [CGR95, Amá07]. Although there has been progress in recent years, the onus of formality does not justify their widespread use; the effort and expertise they require is justified in domains where the cost of software fault is very high, such as the safety-critical niche [CM95].

Visual languages like UML are limited at modelling concerns separately and in isolation [FRG04]. In particular, they lack effective mechanisms to support system-specific concerns; modularisation of *crosscutting* (or non-orthogonal) concerns [THHS99] is not supported. Such research problems are currently being tackled by approaches to Aspect-Oriented Modelling (AOM) [FRG04, WA04, RGF$^+$06, KAK09].

1.2 VCL

This document presents the design of the visual contract language (VCL). VCL tries to address the following problems:

- Enable visual description of aspects not visually expressible using mainstream visual languages such as UML.
- Enhance adoptability of formal techniques by developing formal models using visual descriptions.
- Enhance separation of concerns to tackle complexity of large systems by enabling decomposition of system-specific concerns (such as *crosscutting* concerns, which are typically hard to modularise).

VCL's design presented here includes an approach to behavioural modelling based on *design by contract* [Mey92], and an approach to coarse-grained separation of concerns. It comprises the notations of *package*, *structural*, *constraint*, *behavioural* and *contract* diagrams; each notation constituting a diagram type. VCL's design presented here is accompanied by an outline of its formal semantics. VCL's design is presented and illustrated through a case study.

1.3 Outline

The remainder of this document is structured as follows. Chapter 2 presents the core of VCL. Chapter 3 extends the core VCL with a package mechanism and aspect-orientation. Finally, chapter 4 discusses the results, chapter 5 presents the related work, and chapter 6 presents the conclusions. The chapters given in appendix complete the VCL specification of the case study (chapter A), give the Z that would be generated from this VCL model (chapter B), and present the Z toolkit that is used in the Z generated from the VCL diagrams (chapter C).

Chapter 2

The Core of VCL

This chapter presents the core language of VCL for structural and behavioural modelling. In particular, it shows VCL's ability to describe predicates, which enable the expression of assertions and contracts (made of a pre- and a post-condition) in a modular way. This chapter presents the outline of a syntax and a semantics for VCL. Syntax and semantics of VCL are illustrated using a case study.

In the rest of this chapter, we start by presenting the case study that illustrates VCL. Then, we present the actual syntax and semantics of VCL.

2.1 Running Example

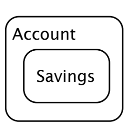
VCL is illustrated here with simple Bank case study, which is also used to illustrate the ZOO semantic domain in [APS05, Amá07]. The case study's requirements are given in table 2.1.

The full VCL model of simple Bank is presented in section A.1. Full Z specification resulting from the VCL semantics outlined here is provided in section B.2.

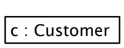
2.2 Syntax of VCL

This section starts by presenting VCL's visual primitives, which are are used in different types of diagrams; they have a core meaning that varies slightly with the context. Next sections then outline abstract syntax of structural, behavioural, constraint and contract diagrams.

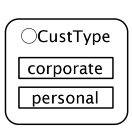
2.2.1 Visual Primitives



VCL *blobs* are labelled rounded contours denoting a *set*. They resemble Euler circles; topological notion of *enclosure* denotes subset relation (to the left, `Savings` is subset of `Account`).



Objects are represented as rectangles; they denote an element of some set. They have a label that includes their name and may include the set to which they belong (e.g. `c` to the left).

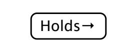


Blobs may also enclose objects, and they may be defined in terms of the things they enclose by preceding the blob's label with the symbol ◯. To the left, `CustType` is defined in this way by enumerating its elements.

R1	The system shall keep information of customers and their Bank accounts. A customer may hold many accounts, but an account is held by one customer.
R2	A customer shall have a *name*, an *address* and a *type* (either company or personal).
R3	A Bank account shall have an account number, a balance indicating how much money there is in it, and its type (either current or savings).
R4	Savings accounts cannot have negative balances.
R5	The total balance of all Bank's accounts must not be negative.
R6	Customers of type *corporate* cannot hold savings accounts.
R7	Customers may open a savings account provided they already hold a current account with the Bank.
R9	The system shall provide an operation to create customers records.
R10	The system shall provide an operation to open bank accounts.
R11	The system shall provide an operation to deposit money onto an account accounts.
R12	The system shall provide an operation to withdraw money from some bank account.
R13	The system shall provide an operation to view the balance of some bank account.
R14	The system shall provide an operation to view a list of all accounts of some customer.
R15	The system shall provide an operation to view a list of all accounts that are in debt.
R16	The system shall provide an operation to delete accounts from the system.

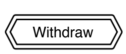
Table 2.1: Requirements of the simple bank system.

Edges connect both blobs and objects. There are two kinds: property and relational. *Property edges*, represented as labelled arrows, denote some property possessed by all elements of the set, like *attributes* in the object-oriented (OO) paradigm (e.g. `balance` to the left).

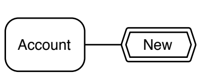


Relational edges are labelled directed lines where direction is indicated by arrow symbol above the line. Their label is within a blob because they define a set of tuples and may be inside blobs. They define or refer to some conceptual relation between blobs (*associations* in OO) – e.g. `Holds` to the left.

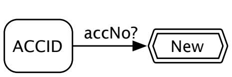
Represented as labelled hexagons, *assertions* (or *constraints*) identify some state constraint or observe (query) operation. They refer to a single state of the system (e.g. `TotalBalIsPositive` to the left).



Contracts are represented as labelled double-lined hexagons. They identify operations that change state; hence, they are double-lined hexagons as opposed to single-lined constraints.



VCL diagrams can include modelling elements from different scopes. *Origin edges* are used to help the reader in identifying the origin of a particular modelling element. They can connect blobs to constraints and contracts. To the left, origin edge indicates that operation `New` is that of blob `Account`.



In constraint and contract diagrams, *communication edges* are used to describe communication constraints involving VCL contracts and constraints. Communication edges are used to say that some object or set of objects are passed to a contract or constraint (e.g. to the left communication edge from blob `ACCID` to contract `New` says that a member of this set, selected non-deterministically, is passed to contract through input `accNo?`).

2.2.2 Structural Diagrams

State structures are defined in a VCL *structural diagram* (SD). Together they constitute an ensemble of structures, defining a state space. VCL model instances are defined by the content of the corresponding model's state structure.

The abstract syntax of SDs is defined in [AK09] using a class metamodel described in Alloy. Briefly, it is as follows:

- A SD is made of a finite number of labelled elements: a *blob*, an *edge*, an *object* or a *constraint*.
- All blobs, relational edges and constraints of a SD have distinct labels. Blobs drawn with a bold line denote a *domain* blob; those drawn with normal lines denote *value* blobs. Domain blobs are part of the state of overall system; they need to be maintained. Value blobs define an immutable set of values; they do not need to be maintained.
- A blob may have blobs and objects *inside*. This inside relation must be acyclic. The label of a blob with things inside may be preceded by symbol ◯ to mean that it is defined by the things it has inside; if the symbol is not present the things inside denote subsets.
- *Property edges* may be drawn between any two blobs that are not inside each other. They define properties of blob at the source end that have as types the blob at the target end. No two property edges with the same source blob have same label. A property edge may have a multiplicity constraint; if not present multiplicity is one; users may specify multiplicities: 1, 0..1 , $*$, or values within a range (e.g. $0\,..\,2$).
- *Relational edges* may be drawn between any two blobs. They define relations between sets. Each end of the edge may have a multiplicity constraint; default value is 1, others are optional, many and range.
- *Objects* define *set elements* when drawn inside some blob; otherwise they define *constants*. A constant must indicate blob to which it belongs. Local constants (connected to some blob) are visible within the blob only; global constants (not connected to any blob) are visible in the scope of the ensemble.
- *Constraints* define invariants. An invariant is *local* when constraint is connected to some blob, and *global* when it is not connected.

Illustration. Fig. 2.1 presents the well-formed SD of simple bank. It is as follows:

- Blobs `Customer` and `Account` are domain blobs. `Customer` has property edges `name`, `cType` and `address`; `Account` has properties `accNo`, `balance` and `aType`.
- Blobs `CustType` and `AccType` are defined by enumeration (symbol ◯); inside, they include all their elements (objects).
- Relational edge `Holds` relates `Customer` and `Account`; multiplicities say that each `Customer` may have many `Accounts`, and that each `Account` has one `Customer`.
- Constraint `SavingsArePositive` is local; all others are global.

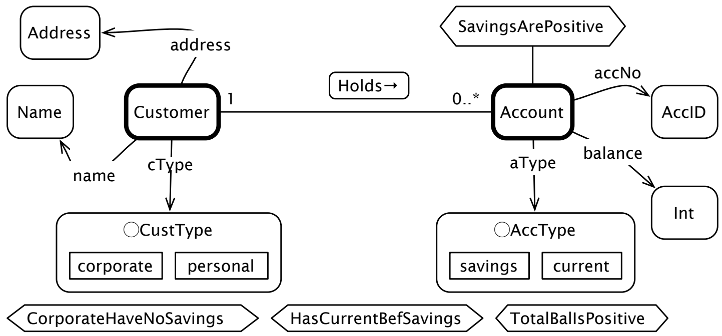


Figure 2.1: Structural diagram of simple Bank

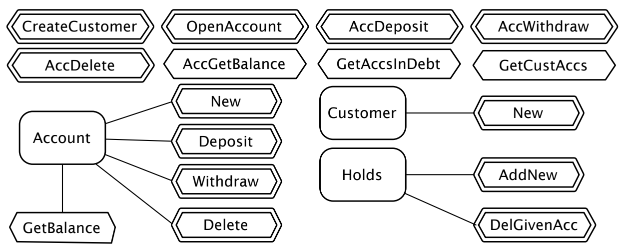


Figure 2.2: Behavioural diagram of simple Bank

2.2.3 Behavioural Diagrams

Operations are VCL's unit of behaviour. They may be *local* or *global*. They are local when they factor some state structure's internal behaviour; global when their context is the overall ensemble of structures. Operations may be further divided into *update* and *observe* (or *query*); the former performs changes of state and the latter performs observations upon the state. A *behavioural diagram* identifies all operations of an ensemble.

BD's syntax is as follows:

- A BD comprises a finite number of *operations* represented as contracts or constraints to denote, respectively, *update* or *observe* operations.
- Operations connected to some blob (representing blob or relational edge from SD) are *local*; those not connected are *global*.

Illustration. A well-formed BD is given in Fig. 2.2. It identifies eight global operations; operations `AccGetBalance`, `GetAccsInDebt` and `GetCustAccs` are observe operations; all other global operations are update operations. BD also identifies several local operations of blobs `Account` and `Customer`, and relational edge `Holds`.

2.2.4 Assertion (or Constraint) Diagrams

A VCL assertion describes a particular condition of some state of the system. They can be used to describe invariants (see [AK09]), and, as this paper illustrates, observe or (query) operations (operations that do not change state).

Abstract syntax of assertion diagrams (ADs) is defined in Alloy in [AK09]. Syntax presented here is a subset of overall syntax (constraint expressions involving logical operators and quantifiers are not included; see [AK09] for further details on this feature). A AD has a *name*, a *declarations* compartment and a *predicate* compartment. Assertions have either a local or global scope; they must have distinct names in some scope.

The declarations compartment comprises:

- A finite number of labelled variables: either objects or blobs. The label is made of the variable's name and its type (blob to which it belongs); no two variables have same name.
- A finite number of imported constraints. Constraint's label comprises an optional up arrow symbol (↑), name of constraint being imported, and an optional rename list. ↑ 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 constraint being imported that are to be renamed (e.g. $[a!/a?]$ says that $a?$ is to be renamed to $a!$).
- Communication edges connecting variables to constraints.

In ADs that describe observe operations, variables may denote communication channels. These are distinguished from ordinary variables through naming conventions: inputs are suffixed with ?; outputs with !.

The predicate compartment may contain a visual expression based on variables (blobs, objects and edges), comprising the following elements:

- A finite number of blobs and objects, which may be connected to other blobs and objects using property and relational edges. Blobs may have other blobs, relational edges and objects inside.
- Property edges are labelled after name of property as defined in SD; in addition, they may include a relational operator in square brackets (e.g. $[\geq]$). A property edge with an object as source refers to the value of property in object; one with a blob as sources refers to the property in all objects of the set.
- A relational edge is labelled with the name of some relational edge defined in SD. They may be used to connect objects and blobs.
- Blobs may have other blobs and relational edges inside, which may mean subsetting (default) or definition (if blob's label is prefixed with symbol ◯). Blobs may be shaded to denote the empty set.

Illustration. Figs. 2.3 and 2.4 give examples of well-formed ADs (from appendix A.1). These are as follows:

Fig. 2.3 presents ADs of local operation `GetBalance` of Account (left) and global operation `AccGetBalance` (right), which promotes this local operation to a global scope. `GetBalance` uses a property edge `balance` of `Account` to connect input `Account` object (`a!`) to output

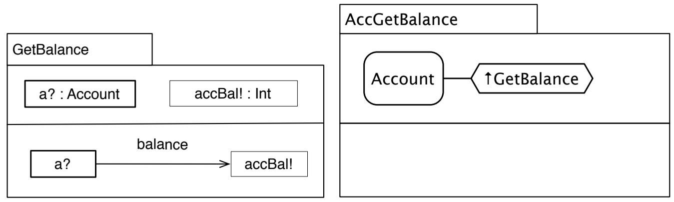


Figure 2.3: Assertion diagrams of operations `Account.GetBalance` and `AccGetBal` (global)

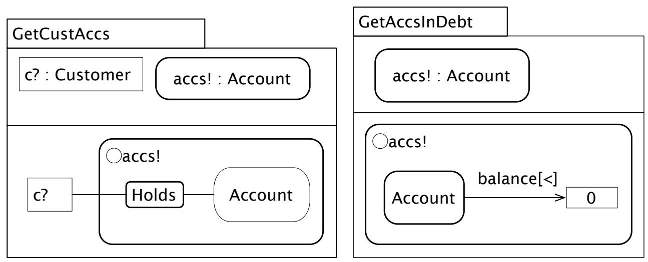


Figure 2.4: Assertion diagrams of global operations `GetCustAccs` and `GetAccsInDebt`

`accBal!` to say that `accBal!` is to hold value of property. Global operation `AccGetBalance` does a total import (symbol $\uparrow$) of the local operation.

Fig. 2.4 presents CntDs of global operations `GetCustAccs` (left) and `GetAccsInDebt` (right). `GetCustAccs` defines output blob `accs!` (symbol ◯) by enclosing relational edge `Holds` and `Account` blob (this obtains range of relation `Holds` restricted on the domain for object `c?`). `GetAccsInDebt` defines output blob `accs!` (symbol ◯) by enclosing blob `Account` and property edge `balance` (this obtains objects of `Account` whose balance is less than 0).

2.2.5 Contract Diagrams

A VCL contract is made of a *pre-* and a *post-condition*. Pre-condition describes what holds before the operation is executed. Post-condition describes effect of the operation, saying what holds after execution.

VCL contract diagrams (CDs) are similar to their constraint counter-parts. Because they involve a pair of states, they comprise two predicate compartments (has opposed to a single predicate compartment in ADs) for pre- and post- conditions. VCL CDs comprise a name, a declarations compartment and a predicate compartment sub-divided into pre- (left) and post-condition (right) compartments. Figs. 2.5, 2.6 and 2.7 present well-formed CDs.

Certain CDs directly express the action of updating state. This is ruled by certain conventions. There are *action* units (object, blob or link), which are identified with a bold line. This action unit can be created, deleted or have its internal state updated; this is described based on a differential semantic interpretation of pre- and post-conditions compartments:

- An action unit on the left compartment but not on the right, means that the unit is deleted.

- An action unit not on the left but on the right, means that the unit is created.

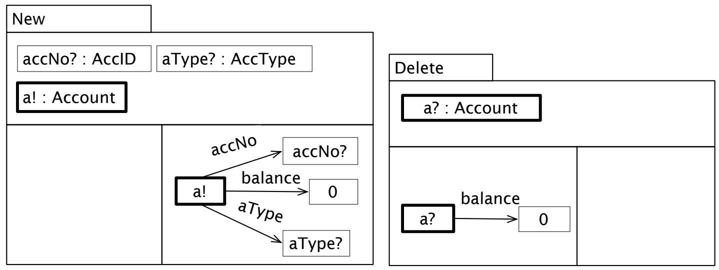


Figure 2.5: Contract diagrams of local operations `New` and `Delete` of blob `Account`

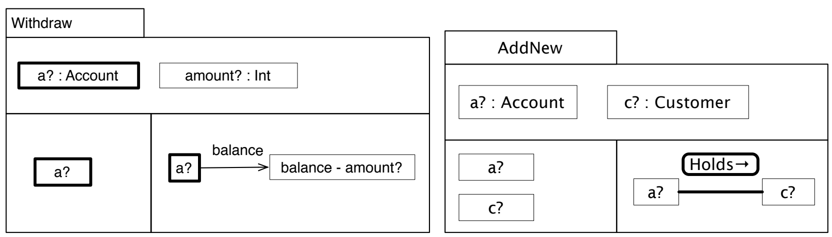


Figure 2.6: Contract diagrams of local operations `Account.Withdraw` and `Holds.AddNew`

- An action unit in both compartments, but with a new value assigned on the right means that the unit is updated.
- A property changes provided right compartment explicitly says so; if right compartment says nothing that means it remains unchanged.

Declarations compartment introduces variables defining the inputs and outputs to the specified operation (inputs are suffixed with ?, and outputs with !), together with the contracts being imported. The syntax is similar to the declarations compartment of CDs, differing in the following:

- Variables representing action units (objects or blobs) are bold-lined.
- Both contracts and constraints can be imported. Imported constraints refer to the before state.
- Communication edges can involve both contracts and constraints.

Syntax of pre- and post-conditions compartment is similar to that of predicate compartment of CDs, differing in the following:

- Action units (object, blob or link) are represented with a bold line.
- pre- and post-conditions compartments may import constraints to strengthen either pre- or post-condition. As CDs do not admit quantified expressions, user may draw separate CDs for more complicated expressions.

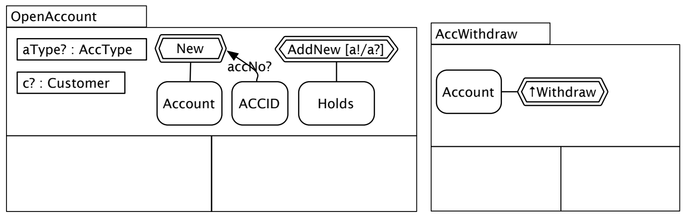


Figure 2.7: Contract diagrams describing global operations `OpenAccount` and `AccWithdraw`

Illustration. ADs of Figs. 2.5, 2.6 and 2.7 are well-formed (from appendix A.1). They are follows:

- Operation `New` (Fig. 2.5, left) declares inputs `accNo?` and `aType?`, and output for action object `a!`. Pre-condition compartment is empty. *Post-condition* gives values to properties of `a!`; `a!` is to be created: it is on the right, but not on the left.
- Operation `Delete` (Fig. 2.5, right) declares action object as input (`a?`). Pre-condition says that action object `a?` must have a balance of 0. Post-condition compartment is empty; `a?` is to be deleted: it is on the left but not on the right.
- Operation `Withdraw` (Fig. 2.6, left) declares two inputs: action object `a?`, and `amount?`. Pre-condition says `a?` exists. Post-condition says that `balance` property of `a?` is given value of expression `balance-amount?` (where `balance` refers to before-state value).
- Operation `AddNew` (Fig. 2.6, right) declares two inputs, `a?`, and `c?`, which are placed un-linked on pre-condition compartment, and linked through relational edge of `Holds` in post-condition; link is to be created as is on the left, but not on the right.
- `OpenAccount` (Fig. 2.7, left) declares inputs `aType?` and `c?`, imports actions of contracts `Account.New` and `Holds.AddNew` (see above), and communication edge from `AccID` to contract `New`. Import of contract `AddNew` includes a renaming: input `a?` of `AddNew` becomes output `a!`.
- `AccWithdraw` (Fig. 2.7, right) does a total import (symbol ↑) of local contract `Account.Withdraw`.

2.3 Semantics of VCL

VCL embodies a *generative* (or *translational*) approach to semantics. It is to be used together with a textual formal specification language, the *target language*, that sits in the background and a target language semantic model. Semantics of a VCL specification is the generated target language specification.

Currently, VCL is given a semantics by mapping diagrams into the ZOO semantic domain [APS05, Amá07], which is a semantic domain of object orientation for the language Z [Spi92, WD96]. We intend to map VCL into other formal languages in the future.

Briefly, semantics of main VCL primitives is as follows:

- A blob is a set. Objects are atoms; members of a set of possible objects that are associated with blob to which they belong.

- Property edges are properties shared by all objects of the set.
- Relational edges are relations between sets.
- An ensemble of state structures is defined as the conjunction of all sets representing blobs and relational edges. Ensembles are used to represent packages and systems. All structures of a SD form an ensemble.
- A constraint describes a condition of a particular state structure or ensemble. It is therefore 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.

The following gives semantics of structural, behavioural, constraint and contract diagrams.

2.3.1 Structural Diagrams

SDs are mapped into ZOO following approach for construction of state spaces outlined in [APS05, Amá07]. Briefly:

- Value blobs that do not have property edges are defined as given sets. Those that are enumerations are defined as free types, and those that have property edges are represented as Z schemas (a record); property edges are represented as fields of the Z schema.
- Domain blobs are defined as a promoted abstract data type [WD96] (a ZOO class). Property edges are represented as fields.
- Relational edges are represented as Z relations.
- Ensemble is formed as conjunction of all Z schemas representing domain blobs and relational edges.
- Constraints identified in a structural diagram are a predicate over a particular state structure (local invariant) or ensemble (global invariant).

Illustration

The following gives ZOO representation of blobs `Name`, `Address`, `AccID`, `CustType` and `Customer`, relational edge `Holds` and state of ensemble defined by SD of Fig. 2.1.

VCL blob `Int` of Fig. 2.1 corresponds to Z primitive set $\mathbb{Z}$ (integers). Blobs `Name`, `Address` and `AccID` are represented as Z given sets:

$$[Name, Address, AccID]$$

Blobs `CustType` and `AccType` defined in VCL by enumeration are defined in Z as free types:

$$CustType ::= corporate \mid personal$$
$$AccType ::= savings \mid corporate$$

Each domain blob has set of all possible objects; existing objects are taken from this set. For this purpose, ZOO defines the set of all possible domain objects:

$$[OBJ]$$

Specific domain objects are subsets of OBJ; these are obtained by using the $\mathbb{O}$ function (see [Amá07] for details).

Blob `Customer` is defined a promoted ADT; this is made of an inner type (schema *Customer*), defining the blob's properties, and an outer type (schema *SCustomer*), which defines set of existing `Customer` objects:

$$\begin{array}{|l}
\hline \textit{Customer} \\
name : Name \\
address : Address \\
cType : CustType \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline \textit{SCustomer} \\
sCustomer : \mathbb{O}\ Customer \\
stCustomer : (\mathbb{O}\ Customer) \nrightarrow Customer \\
\hline
\mathrm{dom}\, stCustomer = sCustomer \\
\hline
\end{array}$$

Relational edge `Holds` is represented as a relation between sets of objects of blobs being related:

$$\begin{array}{|l}
\hline \textit{AHolds} \\
Holds : \mathbb{O}\ Customer \leftrightarrow \mathbb{O}\ Account \\
\hline
\end{array}$$

Overall ensemble of structures that SD of Fig. 2.1 defines is defined by conjoining the definitions of blobs and relational-edges:

$$\begin{array}{|l}
\hline \textit{SystemSt} \\
SCustomer;\ SAccount;\ AHolds \\
\hline
\end{array}$$

Overall system state is constrained by the system's global invariants (see Fig. 2.1); schema representing these are placed in predicate of overall system Z schema:

$$\begin{array}{|l}
\hline \textit{System} \\
SystemSt \\
\hline
CorporateHaveNoSavings \land HasCurrentBefSavings \\
TotalBalIsPositive \\
\hline
\end{array}$$

2.3.2 Behavioural Diagrams

BDs presented here are have two purposes: (a) syntactic sugar, enabling users to have an overview over the functional units of some package, and (b) to set well-formedness rules. BDs assert that certain operations must exist and be defined; they also impose certain visibility rules, which helps in achieving VCL models that are meaningful and well-structured; rules are as follows: (a) in a local scope it is possible to see local operation of associated structure; (b) local operations are not available to the outside world.

2.3.3 On Importing

VCL Importing enables composition of contracts and constraints. Semantically, importing is *conjunction*. When a constraint imports another, the meaning is the conjunction of predicates of importer and imported constraints. Similarly for contracts, importing gives conjunction of pre- and post-conditions of importer and imported contracts. The precise meaning of importing, however, can be controlled as follows:

- VCL provides two means of importing: total and partial. Total importing means that both predicate and variables are imported; as explained in section 2.2.4, this is selected through symbol $\uparrow$. Partial importing (the default mode) means that only predicate is imported; in this case those variables of the imported unit (constraint or contract) that are not declared in the importer unit are hidden.
- In importing, variables are shared or merged when they have the same name. When importer and imported contracts share a variable then the binding involved in the communication does not need to be made explicit. An imported variable is hidden, when its contract is partially imported and it is not declared in the importer contract.
- As explained in section 2.2.4, importing may be subject to renaming of variables in the imported contract. This is used to tune the composition when names of variables differ across units.

2.3.4 Assertion Diagrams

ADs are represented as Z schemas describing a predicate over a particular state structure or ensemble. Semantics of visual expressions of a predicate compartment are as follows:

- An object or blob connected through a property edge to another object or blob is represented as predicate involving a binary operator, which is equality if no user specified operator is provided. A property edge with an object as source is a predicate referring to the object's state; those with a blob as source refer to a set of objects.
- A relational edge denotes a tuple of a relation if it is drawn between objects, and denote domain and range restrictions of the associated relation if there is a blob at one of the ends.
- The inside relation denotes subsetting, unless the label of the enclosing blob is preceded by symbol $\bigcirc$, in which case it denotes equality (or definition).
- When a relational edge is enclosed by some blob, *insideness* may have different interpretations. If only the relation edge is enclosed, that means that the enclosing set is defined as the set of tuples of relation subject to restrictions. The enclosing blob can be defined as the domain and range or relation (subject to restrictions), if relational edge and blob representing either domain or range (respectively) are enclosed.
- Communication edges are represented separately in a Z schema; they state a relation between variables.
- Importing of constraints is subject to rules of importing described in section 2.3.3.

Illustration

Z representation of operation `GetBalance` of Fig. 2.3 (left) is:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccountGetBalance}} \\
\hline
Account \\
accBal! : \mathbb{Z} \\
\hline
accBal! = balance \\
\hline
\end{array}$$

$$SAccountGetBalance == \exists\ Account \bullet \\ \quad \Phi AccountO \land AccountGetBalance$$

This uses an observe promotion schema (see [APS05, Amá07]).

Z definition of `AccGetBalance` of Fig. 2.3 (right) is:

$$AccGetBalance == System \land SAccountGetBalance$$

Z definitions of operations of Fig. 2.4 are as follows:

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{GetCustAccounts}} \\
\hline
System \\
c? : \mathbb{O}\ CustomerCl \\
accs! : \mathbb{P}\ (\mathbb{O}\ AccountCl) \\
\hline
accs! = ran(\{c?\} \lhd holds) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{GetAccsInDebt}} \\
\hline
System \\
acs! : \mathbb{P}\ (\mathbb{O}\ AccountCl) \\
\hline
acs! = \{ac : \mathbb{O}\ AccountCl \mid (stAccount\ ac).balance < 0\} \\
\hline
\end{array}$$

2.3.5 Contract Diagrams

CDs are represented as Z schemas. They define a relation between pairs of states. They are interpreted similarly to ADs, differing in the following:

- They involve a pair of states, rather than a single state. This is expressed in Z using the delta schema convention; the variables of post-condition compartment are primed in resulting Z schema.
- Constraints placed on either pre- or post-condition compartments are composed using Z conjunction.

Illustration

Local operations of Figs. 2.5 and 2.6 are represented in Z as follows:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccountNew}} \\
\hline
\textit{Account}\ ' \\
\textit{accNo}? : \textit{AccID} \\
\textit{aType}? : \textit{AccType} \\
\hline
\textit{accNo}' = \textit{accNo}? \\
\textit{balance}' = 0 \\
\textit{aType}' = \textit{aType}? \\
\hline
\end{array}
\qquad
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccountDelete}} \\
\hline
\textit{Account} \\
\hline
\textit{balance} = 0 \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccountWithdraw}} \\
\hline
\Delta\ \textit{Account} \\
\textit{amount}? : \mathbb{N} \\
\hline
\textit{accNo}' = \textit{accNo} \land \textit{aType}' = \textit{aType} \\
\textit{balance}' = \textit{balance} - \textit{amount}? \\
\hline
\end{array}
$$

$$\textit{SAccountNew} == \exists\ \textit{Account}\ ' \bullet \Phi \textit{BankSAccountN} \land \textit{AccountNew}$$

$$\textit{SAccountDelete} == \exists\ \textit{Account} \bullet \Phi \textit{BankSAccountD} \land \textit{AccountDelete}$$

$$\textit{SAccountWithdraw} == \exists\ \Delta \textit{Account} \bullet \Phi \textit{BankSAccountU} \\ \quad \land\ \textit{AccountWithdraw}$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HoldsAddNew}} \\
\hline
\Delta \textit{Holds} \\
a? : \mathbb{O}\ \textit{AccountCl} \\
c? : \mathbb{O}\ \textit{CustomerCl} \\
\hline
\textit{rHolds}' = \textit{rHolds} \cup \{(a?, c?)\} \\
\hline
\end{array}
$$

Global operations `OpenAccount` and `AccWithdraw` follow ZOO specification of system operations (see [APS05, Amá07]). Operation `OpenAccount` is defined in Z as:

$$\Psi \textit{OpenAccount} == \Delta \textit{System} \land \Xi \textit{SCustomer}$$

$$\textit{OpenAccount0} == [\ c? : \mathbb{O}\ \textit{CustomerCl};\ \Delta \textit{System} \mid \\ \quad c? \in \textit{sCustomer}\ ]$$

$$\textit{ConnAccountNew} == [\ \textit{accNo}? : \textit{ACCID} \mid \textit{accNo}? \in \textit{ACCID}\ ]$$

$$\textit{OpenAccount} == (\Psi \textit{OpenAccount} \land \textit{SAccountNew} \\ \quad \land\ \textit{OpenAccount0} \land \textit{ConnAccountNew} \\ \quad \land\ \textit{AHoldsAdd}[a!/a?]) \setminus (\textit{accNo}?, a!)$$

Above, the two channels of `Account.New` not declared in contract `OpenAccount` (`accNo?` and `a!`) are hidden.

Z definition of operation `AccWithdraw` is:

$$\Psi \textit{AccWithdraw} == \Delta \textit{System} \land \Xi \textit{SCustomer} \land \Xi \textit{AHolds}$$

$$\textit{AccWithdraw} == \Psi \textit{AccWithdraw} \land \textit{SAccountWithdraw}$$

2.4 Conclusions

This chapter outlined the syntax and semantics of the core part of VCL, a language designed for describing structural and behavioural properties of software systems visually and formally. It presented the notations of structural, behavioural, assertion and contract diagrams.

Next chapter extends the core part of VCL with packages and mechanisms to express aspect-oriented like modular compositions.

Chapter 3

Packages and Aspect-Orientation

This chapter extends the core of VCL presented in chapter 2 with VCL's package mechanism. This is done to support coarse-grained separation of concerns and modular composition in an aspect-oriented style. This chapter gives an outline of the syntax and semantics of this extension, which is illustrated with a case study.

In the rest of this chapter, we start by presenting the case study that is used to illustrate the syntax and semantics of the VCL extension presented here. Then, we present the actual syntax and semantics.

3.1 Running Example

The VCL diagrams of figures 2.1 to 2.7 describe the simple Bank case study [APS05] (section A.1). To motivate this paper and illustrate the VCL features introduced here, we introduce the *secure simple bank*, which extends simple bank with the security concerns of *authentication* and *access control*. The new requirements of secure simple Bank are given in table 3.1.

VCL model of this case study is structured around a collection of packages that separate problem domain and security concerns. Full VCL model is given in appendix A. The complete Z specification resulting from this VCL model is given in appendix B.

The secure simple bank case study is to illustrate the VCL package mechanisms in the next sections.

3.2 Extending VCL with Packages

To address coarse-grained separation of concerns in VCL, this paper introduces: (a) the VCL package construct, (b) the package extension mechanisms for building larger packages from smaller ones, and (c) *package diagrams* to define packages.

A package encapsulates state and behaviour. Ordinary packages define a global state; *abstract* packages do not define global state, they act as containers for state structures and their local behaviour to be used in other packages. The global operations of a package constitute the package's interface to the outside world.

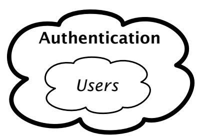
VCL's core (chapter 2) is extended to accommodate the package constructions.This involves a slight extension to VCL's semantic model. The following shows how this is done.

R17	Users are required to authenticate themselves prior to opening a session to use the system, and to close the system's session when no longer need to use the system. This shall be done using the system operations *login* and *logout*.
R18	There are two kinds of users, namely *clerks* and *managers*; different system operations are to be used by one, the other or both. Managers can execute the operations *create customer records*, *open accounts* and *delete accounts*. Clerks can execute operations *deposit* and *withdraw*. Both managers and clerks can execute operations *get balance*, *get customer accounts* and *get accounts in debt*.

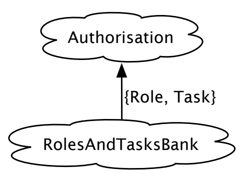
Table 3.1: Requirements of secure simple bank system, which extends the requirements of simple bank (table 2.1).

3.2.1 Visual Primitives

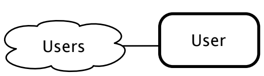
The extension to VCL presented here introduces the following visual primitives.



A VCL model is organised around *packages*, whose symbol is the *cloud*. Packages encapsulate structure and behaviour, and are built from existing ones using *extension*. They are VCL's coarse grained modularity construct. A package extends those packages that it encloses (e.g. `Authentication` to the left).



Override edges are used to describe *override* relationships between packages to say that a package overrides abstract blobs of another in the context of some package definition. Such edges are labelled with the blobs that are being overridden. The override edge to the left, says that package `RolesAndTasksBank` overrides blobs `Role` and `Task` of package `Authorisation`.



Origin edges may include a package as the origin of some modelling element (blob, constraint or contract). To the left, origin edge indicates that blob `User` is from package `Users`.

3.2.2 Package Diagrams, Syntax

In outline, the abstract syntax of package diagrams is as follows:

- A package diagram is made of a finite number of labelled packages; all packages in a package diagram have distinct labels.
- The package being defined is represented with a *bold line*. Packages with their label written in *italics* denote *abstract* packages. Ordinary packages define global behaviour. Abstract packages do not define global behaviour; they act as containers for state structures and their local behaviour to be used by other packages.
- The package being defined may have other packages inside to represent the packages it extends (or incorporats). A package diagram comprises the package being defined and all the packages it extends only. An abstract package cannot extend packages that are not abstract. All the packages being extended must have already been defined.

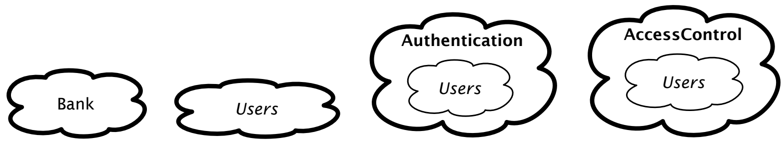


Figure 3.1: Package diagrams defining packages `Bank`, `Users`, `Authentication` and `AccessControl` of secure simple Bank

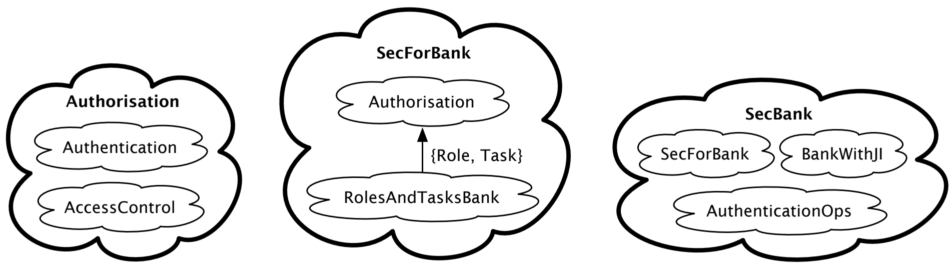


Figure 3.2: Package diagrams defining packages `Authorisation`, `SecForBank` and `SecBank` of secure simple Bank

- The packages being extended may be connected with *override* edges. Override edges define package relations that must be anti-reflexive and anti-symmetric. The edge's label indicates the abstract blobs being defined in the source package; those names must correspond to abstract blobs in the target package, and they must be defined as non-abstract in the source package.

- In general, the global names of structures defined by the packages being extended (names of blobs and relational-edges) must be unique. There are two exceptions to this: (a) there is an overrides relationship between two packages, and in that case the package doing the override may define the blobs being overridden using same name; (b) there is some blob that belongs to the two packages being incorporated and this blob is exactly the same in both packages (in a tool, a user should be warned about this case).

Illustration

Figures 3.1 and 3.2 present well-formed package diagrams defining packages of secure simple Bank; they are as follows:

- Package `Bank` (appendix A.1) encapsulates the problem domain (that is, the model of simple bank presented in chapter 2); it is a self-contained package describing domain of banking (accounts, customers, etc) only (see `Bank`'s SD in Fig. 2.1).

- Package `Users` (appendix A.2) is an abstract package; it defines the `User` blob to be used by other packages.

- Package `Authentication` (appendix A.3) localises the core of the authentication security concern; it extends package `Users`.

- Package `AccessControl` (appendix A.5) localises the core of the access-control security concern; it extends package `Users`.
- Package `Authorisation` (appendix A.6) puts together the authentication and access control concerns. The packages being extended (`Authentication` and `AccessControl`) share blob `User` coming from package `Users`; this is legal because the name denotes a single structure (see above).
- Package `SecforBank` (appendix A.8) customises package `Authorisation` for the purpose of secure simple bank. The package diagram says that package `RolesAndTasksBank` overrides abstract blobs `Task` and `Role` of package `Authorisation`.
- Package `SecBank` (appendix A.11) is the package that represents the overall secure simple bank system. It extends packages `AuthenticationOps` (appendix A.4), so that the system has operations for login and logout, `BankWithJI` (appendix A.10), so that the system has all problem domain functionality encapsulated by package `Bank` with all required security concerns, and `SecForBank`, which configures security for the purpose of secure simple bank.

3.2.3 Package Diagrams, Semantics

A package diagram defines dependency relationships between packages. In outline, the semantics is as follows:

- The extension relationship dictates what packages are visible from the package being defined. Essentially, extension means incorporation: state structures and operations defined in incorporated packages become part of package being defined (the composite package). This is expressed in Z as Z schema conjunction: state of composite package is conjunction of packages being extended and state structures defined in the composite package.
- An abstract package defines sets that may be overriden by some other package. These sets are defined as Z given sets; if overriden this definition is replaced by the one provided by the overriden package.

Illustration

Package `Bank` is represented in Z as ZOO ensemble (see section B.2 for further details). This is defined by conjoining the Z schema definitions of blobs and relational-edges defined in SD of Fig. 2.1; state space of `Bank` would be defined as:

$$\begin{array}{|l} \hline \textit{BankSt} \\ \hline \textit{SCustomer};\ \textit{SAccount};\ \textit{AHolds} \\ \hline \end{array}$$

Overall package state is constrained by the system's global invariants (see Fig. 2.1); schemas representing these invariants are placed in predicate of overall package Z schema:

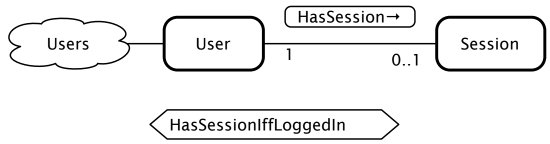


Figure 3.3: Global structural diagram of package `Authentication`

Bank

BankSt

BankCorporateHaveNoSavings
BankHasCurrentBefSavings
BankTotalBalIsPositive

Package `Authorisation` is defined by conjoining the Z schemas corresponding to the packages being extended (see Fig. 3.2):

Authorisation

Authentication; *AccessControl*

Package `Users` does not have a global definition (it is abstract). Package `Authentication` is defined in Z below.

3.2.4 Remaining VCL Diagrams

Abstract Syntax of structural, behavioural, assertion and contract diagrams does not differ substantially from that defined in chapter 2. Essentially, the change concerns being able to use *foreign* modelling elements from the packages being extended; the origin of some foreign element is indicated through an origin edge connected to the package where the element comes from.

3.2.5 Structural Diagrams, Syntax

In outline, the abstract syntax of SDs is extended in the following way:

- SDs may include foreign blobs indicated through an origin edge. Only blobs coming from the packages being extended (as defined in the package diagram) may be included in a SD.
- If an abstract package is extended, then only those structures included in the package's SD are to be part of the package's global state.

Illustration

Figures 3.3 gives the well-formed SD of package `Authentication`. Blob `User` is a domain blob; origin edge indicates that it comes from package `Users` (package `Authentication` extends `Users`, Fig. 3.1). Blob `User` is to be part of state space of `Authentication`. Blob `Session` is another domain blob. Relational edge `HasSession` relates `User`s and their `Session`s.

Figure 3.4: Behavioural diagram of package `Authentication`

3.2.6 Structural Diagrams, Semantics

Semantics of SDs augmented with packages is essentially that introduced in chapter 2. Those blobs coming from abstract packages and that are referenced explicitly in the package's SD become part of the state of the new package.

Illustration

The Z representation of package `Authentication` (package diagram of Fig. 3.1 and SD of Fig. 3.3) is as follows:

$$\begin{array}{|l}\hline \textit{AuthenticationSt} \\ \hline \textit{SUser};\ \textit{SSession};\ \textit{AHasSession} \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{Authentication} \\ \hline \textit{AuthenticationSt} \\ \hline \textit{AuthenticationHasSessionIffLoggedIn} \\ \hline \end{array}$$

The Z representation of domain blob `User` from abstract package `Users` becomes part of the state space of package `Authentication`.

3.2.7 Behavioural Diagrams

Abstract syntax of BDs is extended by allowing the inclusion of local operations for foreign blobs to enable extension of local behaviour for foreign blobs and relational-edges.

BDs do not add anything in terms of semantics. They are, essentially, syntactic sugar, imposing constraints on what operations can be defined, and how they can be defined. These rules are as defined in chapter 2.

Illustration

Package `Users` just defines `User` blob to represent system users. This blob is then used in different ways in packages `Authentication` and `AccessControl`. Figure 3.2.7 presents the well-formed behavioural diagram of package `Authentication`. This includes a local operation of foreign blob `User`: an observe operation to indicate whether a user is logged-in the system or not.

3.2.8 Assertion and Contract Diagrams, Syntax

Abstract syntax of ADs and CDs is extended as follows:

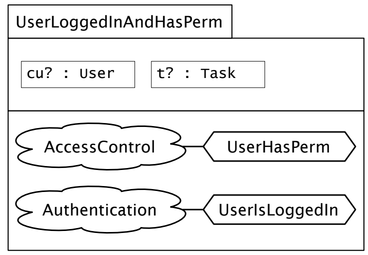


Figure 3.5: Assertion diagram describing observe operation `UserLoggedInAndHasPerm` of package `Authorisation`

- ADs may import foreign observe operations and refer to foreign blobs coming from the packages being extended. Foreign entities must come from the packages being extended, and are indicated using the origin edge. Only those operations that are global can be used as foreign (local definitions of a package are not visible to the outside world).
- CDs may import foreign operations (update or observe) and refer to foreign blobs. Rules regarding use of foreign entities is the same as ADs.

Illustration

Package `Authentication` defines authentication data; it is possible to know which users are logged-in. Package `AcessControl`, on the other hand, defines permission assignments, saying which uses are allowed to execute certain tasks. These two concerns are put together in package `Authorisation`. The well-formed AD of Fig. 3.5 describes the observe operation `UserLoggedInAndHasPerm`, which says whether a user is both logged-in and has the required permissions to execute some task (this is a crosscutting operation); the definition in the AD just puts together the observe operations `UserHasPerm` of package `AccessControl` and `UserIsLoggedIn` of `Authentication`.

3.2.9 Assertion and Contract Diagrams, Semantics

Semantically, not much differs in this extension for assertion and contract diagrams. Global package operations are built using Z schema conjunction by conjoining the foreign operations, the operation's frame and the pre- and post-condition predicates of the composite operation. The aspect of the Z representation that requires further work is the definition of the operation's frame; the frame of the foreign operation needs to take into account the new context in which the operation is used (to avoid the frame problem [BMR95]). We see how this is done in section 3.4.2.

Illustration

The Z representation of the observe operation described in AD of Fig. 3.5 is as follows:

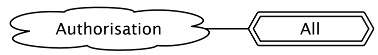


Figure 3.6: Behavioural diagram of package `SecForBank` making use of integral extension.

$$\begin{array}{|l}
\hline
\quad \textit{AuthorisationUserLoggedInAndHasPerm} \\
\textit{Authorisation} \\
\textit{AccessControlUserHasPerm} \\
\textit{AuthenticationUserIsLoggedIn} \\
\hline
\end{array}$$

Here, this Z schema is formed as the conjunction of state of package `Authorisation` (the constraints of the composite package need to be taken into account in the composite operation; the frame of observe operations is the state itself, as nothing changes), and operations `UserHasPerm` of package `AccessControl` and `UserIsLoggedIn` of `Authentication`.

3.3 Aspect Orientation in VCL

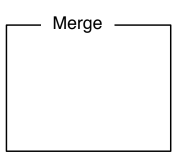
Section 3.2 introduced VCL's package construct, which enhances the modularity of VCL by providing a mechanism for coarse-grained separation of concerns. This fits nicely into VCL's core semantic model outlined in chapter 2. In terms of state, packages are just ensembles (represented in Z as Z schemas) that can be composed to build larger packages; package operations are also defined using Z schema conjunction: the operations being extended are conjoined with the pre- and post- condition predicates that the composite operation defines as its own. This provides an approach to *incremental definition* based on packages, where a new package is built from an existing one by adding more state and behaviour.

However, these incremental definitions have to be customised to each case. They enable aspect-oriented like compositions, but are not as modular as we would like them to be. As we will see, we have the need of applying the same modular construction to a group of operations.

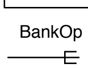
We now see how we can go from the basic package mechanism introduced in section 3.2 to a more aspect-oriented form of package composition. All these forms of composition are described in VCL behaviour diagrams.

3.3.1 Visual Primitives

To refer to all operations of some package, we introduce the *all* qualifier. This consists of a contract labelled with keyword *All* connected with an origin edge to some package; this refers to all operations of the package. An example of a all qualifier can found in Fig. 3.6.



To refer to a group of operations to apply a particular operation on them, VCL uses *boxes* (rectangles), with a label at the top to indicate the kind of box. Operations are placed inside the box. There are two kinds of boxes: *merge* or *join*. Box to the left is a merge box.



To represent the fact that a behaviour is being introduced to a group of operations, VCL uses the labelled *fork edge* (e.g. fork edge to the left). Fork edges are used to connect a contract to a join box, to say that the extra behaviour of the contract is to be added to the group of operations inside the box; label indicates a name that is used as a representative of the operations inside the box.

3.4 Integral Extension

A package encapsulates state and behaviour. When packages are incorporated, the state is also incorporated (if package is not abstract), but not the operations; these operations can be used to define new operations in the composite package but are not made available to the outside world. Integral extension promotes foreign operations to package operations in composite package (they become available to outside world). Operations defined this way are used in the new context integrally (they are not changed in any way).

3.4.1 Syntax

Syntax of BDs introduced in chapter 2 is extended with integral extensions. In outline, abstract syntax is as follows:

- BDs may include foreign operations (contracts or assertions) connected with an origin edge to the package where they come from. Only foreign operations that are global may be included in BDs.
- It is often the case that one wants to include in a BD all global operations of some package. This is described using the all qualifier visual primitive (see above).
- All global operations of a package must have unique names (as defined in chapter 2); hence, name clashes resulting from the inclusion of foreign operations in a BD (e.g. two operations with same name, but coming from different packages) constitutes a well-formedness error.

Illustration

Package `SecForBank` of secure simple bank (Fig. 3.2) customises package `Authorisation` for the specific purpose of secure simple Bank. Operations of `Authorisation` are to be used in the context of `SecForBank` unaltered, but must take the constraints introduced by the customisation into account. This is expressed using an integral extension in the well-formed BD of Fig. 3.6, which says that all operations of package `Authorisation` are to be used unaltered in the context of package `SecForBank`.

Another well-formed BD containing an integral extension is that of package `SecBank` in figure 3.8.

3.4.2 Semantics

Operations defined using integral extensions are represented in Z following the general outline of package operations discussed n section 3.2.9. That is, they are described in Z using Z schema conjunction: by conjoining foreign operation and the operation's frame.

The frame is required so that new context (the composite package) is taken into account in the new operation's definition (this is done to avoid the frame problem [BMR95], see [APS05]). The frame's definition involves using Z schema projection to say that the state of the composite package that is not part of the state of the foreign operation's package does not change.

Illustration

We show how operations resulting from integral extensions of Figures 3.6 and 3.8 would be represented in Z.

Integral extension of Fig. 3.6 defines a single operation: observe `UserLoggedInAndHasPerm`. The new operation formed by integral extension is the conjunction of the foreign operation and the package schema:

$$\begin{aligned}&\mathit{SecForBankUserLoggedInAndHasPerm} == \mathit{SecForBank} \land\\&\quad \mathit{AuthorisationUserLoggedInAndHasPerm}\end{aligned}$$

Update operations require the definition of the operation's frame. Frame of operations resulting from integral extension of Fig. 3.8 is defined as:

$$\mathit{CommonWithAuthenticationOps} == \mathit{SecBank} \upharpoonright \mathit{AuthenticationOps}$$

$$\mathit{SecBankWithoutAuthenticationOps} == \exists\, \mathit{CommonWithAuthenticationOps} \bullet \mathit{SecBank}$$

$$\Psi \mathit{SecBankOpsFromAuthenticationOps} == \Delta \mathit{SecBank} \land \Xi \mathit{SecBankWithoutAuthenticationOps}$$

The operation `Login` resulting from the integral extension of Fig. 3.8, is defined as the conjunction of the frame and foreign operation:

Login

$$\begin{array}{|l}\Psi \mathit{SecBankOpsFromAuthenticationOps}\\ \mathit{AuthenticationOpsLogin}\end{array}$$

3.5 Merge Extension

VCL's packaging mechanism provides a modularity construct to support separation of concerns. This enables the separate specification of related behaviours, each corresponding to a particular concern. To then join related behaviours, VCL introduces the *merge extension*.

3.5.1 Syntax

In outline, abstract syntax of BDs introduced in chapter 2 is extended in the following way:

- BDs may include one merge box, which has inside a finite set of foreign operations (at least one). All the operations included in the merge box with the same name are merged into a new package operation that joins the two behaviours. It is not possible to merge update operations with observe ones.
- All the operations included in the merge box that do not have a matching operation (another operation with same name) are ignored (in a VCL tool a warning should be given to the user regarding this).

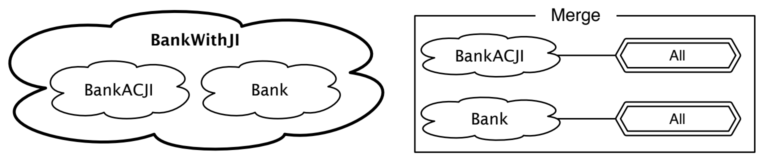


Figure 3.7: Package and Behavioural diagram of package `BankWithJI`. Behavioral diagram defines operations of package using *merge extension*.

Illustration

Operations of package `Bank` (such as `Withdraw`) need to be protected with an access-control mechanism as defined by the requirements (section 3.1). This must involve composition between what is defined in package `Bank` and what is defined in package `AccessControl`.

Figure 3.7 defines the package and behavioural diagram of package `BankWithJI` of secure simple Bank. This package augments the domain package `Bank` with a surrounding interface, defined in package `BankACJI`, to enable composition with package `AccessControl`[1].

BD of Fig. 3.7 is well-formed and makes use of *merge extension*. It says that all operations from package `Bank` are to be merged with all operations from package `BankACJI`; operations with same name are conjoined into a single operation for package `BankWithJI`[2].

3.5.2 Semantics

Semantics of *merge extension* is Z schema conjunction. The pair of operations being merged is conjoined to form a new package operation that performs the behaviour of both operations. As with integral extension, the frame of the operation needs to be defined: the new state introduced by the composite package must not change.

Illustration

To represent in Z, the result of the merge extension defined in Fig. 3.7, we need to define the frame for all update operations. We start by describing a Z schema made of the variables that are defined in package `BankWithJI` but not in `Bank`.

$$
\begin{aligned}
&\mathit{CommonWithBank} == \mathit{BankWithJI} \upharpoonright \mathit{Bank}\\
&\mathit{BankWithJIWithoutBank} == \exists\, \mathit{CommonWithBank} \bullet \mathit{BankWithJI}
\end{aligned}
$$

From this schema, we build the frame of the operation defined by the merge extension. The frame says that the state defined in `BankWithJI` but not in `Bank` must not change.

$$\Psi \mathit{BankWithJIMergeOps} == \Delta \mathit{BankWithJI} \wedge \Xi \mathit{BankWithJIWithoutBank}$$

Then, a package operation is defined for each merge by conjoining the frame and the operations being merged. Operation `CreateCustomer` is defined as follows:

[1] Package `BankACJI` says for each operation of `Bank` what is the corresponding *AccessControl* task.

[2] Package `BankWithJI` contains a set of global operations whose set of names is the same as the set of names of global operations in package `Bank`

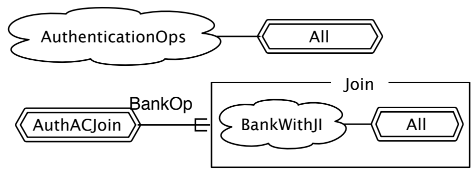


Figure 3.8: Behavioural diagram of package `SecBank`. Operations are defined using *join extension*.

BankWithJICreateCustomer
Ψ *BankWithJIMergeOps*
BankCreateCustomer
BankACJICreateCustomer

Remaining operations would be similarly defined.

3.6 Join Extension

We have seen that it is possible to separate and localise cross-cutting concerns, such as *authentication* and *access-control*. To enable composition of crosscutting concerns, VCL provides the join extension. Join extension allows a certain extra behaviour to be introduced (or joined) to a group of operations; this enables the composition of cross-cutting concerns. This extra behaviour is expressed as a VCL contract describing a pre- and a post-condition.

3.6.1 Syntax

In outline, abstract syntax of BDs introduced in chapter 2 is extended with join extensions as follows:

- A BD may include one or more *join boxes*, which have inside a finite set of operations (at least one).
- One or more *join contracts* are connected with fork edges to each *join box*; this describe the extra behaviour to be introduced onto the set of operations inside the box. The edge is labelled with a name that represents any of the operations inside the box in description of join contract.

Illustration

In the VCL model presented so far, domain concern of banking and security concerns of authentication and access control are all described separately. These separate concerns must be woven together in the package `SecBank`, which represents the secure simple bank system. This composition must take into account a crosscutting behaviour, which affects all domain operations of `Bank`: each user performing the operation must be authenticated in the system and must be allowed to execute the operation.

Package `BankWithJI` equips domain package `Bank` with an interface to enable the composition of this crosscutting behaviour. Package `SecBank` performs this composition using a join

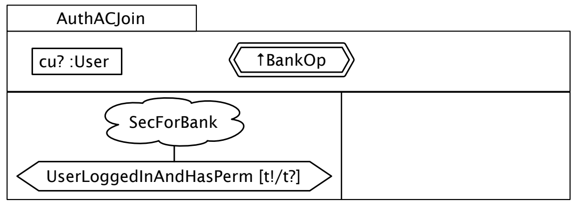


Figure 3.9: Contract diagram for join contract `AuthACJoin` in package `SecBank`.

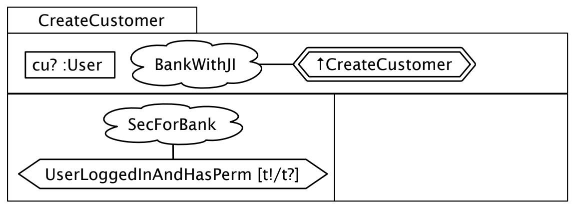


Figure 3.10: Contract diagram that would result from the join extension of join contract `AuthACJoin` with operation `CreateCustomer`.

extension in the well-formed BD of Fig. 3.8; it says that all operations of package `BankWithJI` are to be joined with the extra behaviour described in contract `AuthACJoin`.

Join contract `AuthACJoin` is described in Fig. 3.9. It declares an input object `cu?` representing the current user of the system and includes the constraint `UserLoggedInAndHasPerm` in the pre-condition. This requires that the current user ($cu?$) is logged in and has permissions corresponding to the `BankOp` contract (which represents one of the operations inside the join box); the `BankOp` issues an output identifying the system task to execute (variable $t?$). Figure 3.10 presents the result of the join extension for the operation `CreateCustomer`.

3.6.2 Semantics

Semantics of join extension is Z schema conjunction. The contract representing the extra behaviour is conjoined with each operation inside the join box to form new operations named after the ones inside the box. As with integral and join extensions, the frame of the operation meeds to be defined: the state that does not affect the operation must not change.

Illustration

To represent in Z the result of the join extension defined in Fig. 3.8 and the join contract of Fig. 3.9, we define the frame of all update operations affected by the join extension:

$$CommonWithBankWithJI == SecBank \upharpoonright BankWithJI$$

$$SecBankWithoutBankWithJI == \exists\, CommonWithBankWithJI \bullet SecBank$$

$$\Psi SecBankOpsFromBankWithJI == \Delta SecBank \land \Xi SecBankWithoutBankWithJi$$

The frame is used to define al update operations. The integral extension for operation `CreateCustomer` would result in the following Z definition:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CreateCustomer}} \\
\hline
\Psi \mathit{SecBankOpsFromBankWithJI} \\
cu? : \mathbb{O}\, \mathit{UserCl} \\
\mathit{BankWithJICreateCustomer} \\
\hline
\mathit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\
\hline
\end{array}
$$

This is what would also result from the `CreateCustomer` contract of Fig. 3.10.

3.7 Conclusions

This chapter outlined the syntax and semantics of the VCL package construct and associated aspect-oriented composition mechanisms. This has been illustrated using the secure simple bank case study that is made of several orthogonal concerns. This chapter showed how the VCL mechanisms introduced in this chapter were effective at coarse-grained separation of concerns, and aspect-oriented modular composition.

Chapter 4

Discussion

This paper presents the design of VCL, outlining its formal syntax and semantics. This section discusses the results presented here at the light of several criteria that is relevant for modelling of software systems.

4.1 Design of VCL and VCL's approach to modelling

This paper presents the design of VCL, outlining VCL's diagram types and their syntax and semantics. VCL is designed for abstract specification at the level of requirements (or high-level designs). This paper presents VCL's approach to structural and behavioural modelling, and coarse-grained separation of concerns.

A VCL model is organised around *packages*, which are reusable units encapsulating structure and behaviour. Packages enable decomposition and localisation of concerns and are composable. Larger packages are built from smaller ones using VCL's *package extension* mechanism. Packages are defined in *package diagrams*. Ultimately, there is a system package that defines overall state space and behaviour of the system, which is composed of several subpackages representing the various system concerns.

A package encapsulates a set of state structures, which define the package's state space. This is defined in the package's *structural diagram*. VCL model instances are defined by content of model's state structures.

Operations are VCL's unit of behaviour. They either observe or change the content of state structures and are specified in constraint and *contract diagrams*. Operations may be *local* or *global*. They are local when they factor some state structure's internal behaviour. They are global when their context is the overall package; they involve a collection of state structures. A *behavioural diagram* identifies all operations of a package.

The semantic model of VCL illustrated here, ZOO [APS05, Amá07], is well studied. It has been applied to other case studies, such as library system [ASP04], invoice processing system [APS06] and part of the immune system [APZ08]. In these applications, ZOO was used together with UML class and state diagrams.

VCL's formal semantics outlined here was part of the process of designing and experimenting the language. Many features of VCL were obtained by abstracting the structures generated by ZOO. However, although designed with Z and ZOO in mind, VCL is more general providing a set of visual primitives to express structures and concepts that are independent from their various mathematical representations. We are experimenting VCL with the Alloy formal language. Full formal Z semantics of VCL model used here is given in appendix B.

4.2 Modularity and Composition

This paper highlighted VCL's modularity. VCL contracts and constraints are pieces that can be used in multiple contexts. This enables separation of concerns at the level of specification of behaviour; local operations are specified separately and independently from global ones and composed to form many global behaviours. Contract compositions illustrated here involve conjunction only, but, it is possible to use disjunction and negation.

VCL packages enable decomposition of a problem into meaningful and manageable pieces addressing system-specific concerns. VCL can be classified as a *symmetric* AOM approach: crosscutting and non-crosscuting modules all are represented equally as VCL packages that can be composed using VCL's compositions mechanisms.

VCL supports a *plug-and-play* style of composition, which is what gives AOM the edge in terms of what can be decomposed modularly. Modules realising concerns of interest can be plugged-in or plugged-out in response to changing requirements, which facilitates evolution. VCL's compositions are additive and non-invasive (composed packages are not changed); they describe how packages are plugged in or out. *Plug-and-play* composition in VCL, however, usually involves a configuration step. This configuration is, in most cases, trivial; it involves integral, join and merge extensions, which have nice modular properties. More complicated configurations involve custom extensions.

In the VCL model presented here, authentication and access control were added to a collection of packages representing the problem domain concerns in a non-invasive way (that is, composed packages were not changed), where the laborious part consisted of defining the surrounding interface between the packages being composed. Once this was done, the composition itself was trivial; it involved *merge* and *join* extension. In other cases, the composition involves a custom extension, which, although more laborious, can also be specified in a modular and non-invasive way.

4.3 Usability

Usability has been a concern throughout VCL's design. VCL has been designed to be well matched to meaning and to enable users to infer meaning from patterns, following usability guidelines [GP96, GT00, Moo09]. The emphasis on modularity is also because modularity helps to reduce the cognitive overload [Moo09]:

- VCL's visual concepts are designed to be well-matched to meaning and give good sense of their mathematical underpinnings. VCL's blob symbol, a circular contour denoting a set, has been chosen because it is a well-known mathematical visual concept (as Venn or Euler circles). The *cloud* symbol to represent packages has been chosen because it alludes to a world of its own. The declarations, pre-conditions and post-condition compartments of contract diagrams closely mimic underlying structure of operations. We have designed the *fork edge* with a fork shape at one of its end to mean that a behaviour (usually the crosscutting behaviour) is to be introduced to the operations within the box.

- To enable users to infer meaning from patterns, VCL comprises a minimal set of primitives that have some core meaning, which varies slightly with the context (*consistency* guideline of [GP96]). Blobs define given sets in structural diagrams, and derived sets or set references in constraint diagrams. Objects denote a specific object of a set; depending on context, it may define a variable, constant or expression. Package extension construction uses the *insideness* spatial property, which is also used with blobs; the same insideness spatial property is used in join and merge boxes to mean that a join or a merge is to be applied to the operations inside the box.

- To reduce search for modelling items, achieve coherent groupings and integrate sources of information (following guidelines in [LS87, CMS99]), VCL is designed so that its constituent parts make a coherent and integrated whole. SDs, for instance, provide an integrated definition of a system structure together with their constraints; constraints are then defined in other diagrams.

- VCL stands out from other approaches to contracts because of its modular capabilities (see chapter 5). This is known to help to reduce the cognitive overload [Moo09].

4.4 Aspect-orientation

Both authentication and access-control concerns are crosscutting. This paper showed that VCL was capable of describing a composition of these concerns with the domain concerns to build a model of the overall system. This illustrated many uses of the VCL package techniques presented here. First, we've built a surrounding interface to enable aspect composition with the domain package `Bank`; this involved defining the interface package `BankACJI` and then compose it with the domain package `Bank` using a *merge extension* to make package `BankWithJI` (Fig. 3.7). Then the crosscutting behaviours were woven into the overall system by using *join extension*.

4.5 Verification and validation

VCL is designed with a formal semantics. VCL models are translated into a Z specification[1]. These Z specifications are key for verification and validation in VCL because they enable formal validation (or testing) and verification of certain desired properties. In Z, this is done using theorem proving.

A visual approach to formal validation of UML-based models that have a ZOO semantics (also used for VCL), called *snapshot analysis*, is developed in [ASP04, Amá07]. This tests a Z specification against examples and counter-examples (represented as snapshots) using theorem proving. We intend to incorporate this approach in VCL in the future.

4.6 Reuse

Security concerns of authentication and access control are recurring concerns in today's software systems. VCL packages presented here that encapsulate them are potentially reusable.

[1] Generation of Z from VCL is an ongoing effort at time of writing; the generation of Z from VCL for the VCL model of secure simple bank presented here is illustrated in appendix B.

	Total Lines of Z	From visual	Percentage of visually
With VCL	490	484	98.8%
With UML as in [Amá07, APS07]	439	195	44.4%

Table 4.1: Visual expressiveness in relation to generated Z: VCL vs UML in a model the simple Bank system.

In fact, the packages of the *authentication* concern presented here were also used in the large case study modelled in [AMGK09]; package `AccessControl` slightly differs from that used in [AMGK09].

To enhance reuse, VCL packages should be made as generic as possible and adaptated, through a customisation, to a context. This is done through a configuration in an additive and non-invasive fashion. This can be observed in the customisation of generic package `Authorisation` to context of secure simple bank that resulted in package `SecForBank` (Fig. 3.8).

4.7 Scalability

VCL's modularity features help designers to master complexity of large problems. They separate and localise concerns, which facilitates understanding, and enable their modular composition. Case study used here is not large – it is made of three concerns –, but VCL presented here has been applied successfully to a substantially larger case study in [AMGK09].

4.8 Visual Expressiveness

Unlike UML, VCL contracts specify behaviours totally. UML behavioural descriptions are partial. UML sequence and collaboration diagrams describe scenarios (or traces); UML state diagrams describe state transitions of components as a whole, hiding behaviour behind actions (usually complemented with OCL).

VCL described all eight system operations of package `Bank`. UML-based specification of [APS05, Amá07] needed to resort to Z to totally describe them. Table 4.1 compares VCL model of `Bank` package presented here with UML-based model of [APS05, Amá07] in relation to generated Z. VCL gives a 54.4% increase in terms of what is expressed visually. The 1.8% that could not be expressed visually corresponds to invariant `TotalBalIsPositive` (see Fig. 2.1) that could not be expressed visually and was expressed directly in Z by embedding the Z text into a AD (see section A.1).

VCL contract diagrams notation is designed to describe simple pre- and post-conditions and compositions of local operations. It does not support quantification directly. For more involved pre- or post-conditions that require quantification, user may draw a constraint diagram and then place it in either the pre- or post-condition compartment. In the application of VCL to a large case study [AMGK09], we did not require quantification to express contracts.

All concerns and compositions required to model the secure simple Bank case study were expressed visually in VCL using the coarse-grained package mechanism introduced here.

4.9 Practical Value

VCL is visual and modular for practical reasons: visual representations have proved valuable in engineering [Fer77], and modularity helps tackling complexity. VCL was applied successfully to a large case study [AMGK09]; we found:

- VCL packages effectively separated system-specific concerns; plug-and-play composition facilitated construction of composite packages.
- It was more productive to specify in VCL than in Z directly; visual nature of VCL enhanced usability, readability and communication of resulting model.

We intend to use [AMGK09] to produce further empirical evidence. Currently, we are developing tool support for VCL[2] to further enhance VCL's practical value.

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

Chapter 5

Related Work

Use of left and right compartments to mean pre- and post-conditions are inspired by Catalysis' snapshot-pairs [DW98]. In Catalysis, each snapshot represents a specific system state. Meaning of VCL contract diagrams is an operation specification (a relation between before and after states).

Some approaches augment UML with contracts described as object-diagram pairs. Engels et al [LSE05, ELSH06] translates, using graph transformation, UML class diagrams and contracts to Java skeletons and JML assertions. Visual OCL [EW06] also uses graph transformations to go from contracts to OCL. Like VCL, these approaches follow a translational approach to semantics. Constraint diagrams [FFH05, HSS09] notation has many similarities with VCL; it describe behaviour based on pre- and post-conditions and uses circles to represents sets and *insideness* to represent subset relationship. Unlike VCL, this approach does not take a translational approach to semantics; instead, the language is given a semantics to enable modelling and reasoning at the visual level. Constraint diagrams, however, is a formally defined notation; VCL presented here is a design of a language with an outline of a formal semantics. VCL's design presented here, however, provides better modularity mechanisms than all these approaches to contracts: modularity of contracts, and coarse-grained modularity based on packages.

Several aspect-oriented modelling (AOM) approaches enhance UML to support modularisation of crosscutting concerns [FRG04, RGF$^+$06, WA04, KAK09]. France et al [FRG04, RGF$^+$06] is an approach to architectural modelling based on class models and textual OCL operations; template-based aspectual models are instantiated for a particular context and then composed with a base model; result of composition is a merge based on signatures (a conjunction). VCL differs in that it targets requirements modelling, it is entirely visual, and does not use templates to describe aspect models. Composition mechanism of [FRG04] are akin to VCL's merge extension.

Other AOM approaches represent crosscutting behaviour as scenarios [WA04, KAK09]. Whittle and Araújo [WA04] build *aspectual scenarios* of crosscutting behaviour as interaction templates; these are composed with base scenarios (described as UML sequence diagrams) through an integration operator to synthesise state diagrams. Kienzle et al [KAK09] define aspect models made of class, state and sequence diagrams; user must identify *pointcuts* to insert crosscutting behaviour; behavioural models are then composed based on a weaving algorithm. VCL differs from these approaches in that it uses contracts to totally describe behaviour (as opposed to partial descriptions), and does not require complex synthesis or weaving algorithms – compositions are done at level of semantics, hidden from visual world, and they involve simple conjunction with some merging of names. Unlike [WA04], VCL does not need ordering constraints in specification of integration for crosscutting behaviour; at VCL's level of abstraction computations of components occur in parallel and usually composition involves a simple conjunction (of pre- and post- conditions). Unlike [KAK09], VCL does not use pointcuts; operations are specified separately and independently, and then joined in many contexts using join extension.

Chapter 6

Conclusions and Future Work

This paper presents the design of VCL, a visual and formal language for modular abstract specification of software systems, outlining the language's syntax and semantics. This paper presents design of structural, behavioural, assertion, contract and package diagrams, together with the mechanisms of composition of VCL packages. VCL's design presented here has been illustrated with a case study. Full VCL model of case study is given in appendix A; the full Z specification resulting from the VCL model is given in appendix B. We are currently formalising VCL's syntax and semantic mapping, and developing VCL's tool.

Most relevant contribution of VCL's design presented here is its modular approach to modelling. VCL's contracts and assertions have good modular properties; they enable the description of larger constraints and operations from smaller ones. The package mechanism and the associated mechanisms for aspect-oriented modular composition, enable an effective coarse-grained separation of concerns; system specific-concerns can be plugged-in or out of a system. To our knowledge, no other modelling language that takes a visual approach to design by contract achieves this level of modularity.

References

[AHGT06] Bente Anda, Kai Hansen, Ingolf Gullesen, and Hanne Kristin Thorsen. Experiences from introducing UML-based development in a large safety-critical project. *Empirical Software Engineering*, 11(4):555–581, 2006.

[AK09] Nuno Amálio and Pierre Kelsen. The abstract syntax of structural VCL. Technical Report TR-LASSY-09-02, University of Luxembourg, 2009. available at `http://vcl.gforge.uni.lu/doc/abs-syn-strt-vcl-report.pdf`.

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

[AMGK09] Nuno Amálio, Qin Ma, Christian Glodt, and Pierre Kelsen. VCL specification of the car-crash crisis management system. Technical Report TR-LASSY-09-03, University of Luxembourg, 2009. available at `http://vcl.gforge.uni.lu/doc/vcl-cccms.pdf`.

[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.

[APS06] Nuno Amálio, Fiona Polack, and Susan Stepney. UML+Z: Augmenting UML with Z. In Henri Abrias and Marc Frappier, editors, *Software Specification Methods*. ISTE, 2006.

[APS07] Nuno Amálio, Fiona Polack, and Susan Stepney. Frameworks based on templates for rigorous model-driven development. *ENTCS*, 191:3–23, 2007.

[APZ08] Nuno Amálio, Fiona Polack, and Jing Zhang. Autonomous objects and bottom-up composition in ZOO applied to a case study of biological reactivity. In *ABZ 2008*, volume 5238 of *LNCS*, pages 323–336. Springer, 2008.

[ASP04] Nuno Amálio, Susan Stepney, and Fiona Polack. Formal proof from UML models. In *Proc. ICFEM 2004*, volume 3308 of *LNCS*, pages 418–433. Springer, 2004.

[BMR95] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions in Software Engineering*, 21(10):785–798, 1995.

[CGR95] Dan Craigen, Susan Gerhart, and Ted Ralston. Formal methods reality check: industrial usage. *IEEE Transactions on software engineering*, 21(2):90–98, 1995.

[CM95] George Cleland and Donald MacKenzie. Inhibiting factors, market structure and industrial uptake of formal methods. In *WIFT'95*, pages 46–60. IEEE, 1995.

[CMS99] Stuart K. Card, Jock D. Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, 1999.

[DW98] Desmond D'Souza and Alan Cameron Wills. *Objects, Components and Frameworks with UML: the Catalysis approach*. Addison-Wesley, 1998.

[EFLR98] Andy Evans, Robert B. France, Kevin Lano, and Bernhard Rumpe. The UML as a formal modelling notation. In *UML'98*, volume 1618 of *LNCS*, pages 336–348, 1998.

[ELSH06] Gregor Engels, Marc Lohmann, Stefan Sauer, and Reiko Heckel. Model-driven monitoring: an application of graph transformation for design by contract. In *ICGT 2006*, 2006.

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

[Fer77] Eugene S. Ferguson. The mind's eye: nonverbal thought in technology. *Science*, 197(4306):827–836, 1977.

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

[FRG04] R. France, I. Ray, and S. Ghosh. Aspect-oriented approach to early design modelling. *IEE Proc. Softw.*, 151(4):173–185, 2004.

[GP96] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *J. of Visual Languages and Computing*, 7(2):131–174, 1996.

[GT00] Corin Gurr and Konstantinos Tourlas. Towards the principled design of software engineering diagrams. In *ICSE 2000*, 2000.

[HSS09] John Howse, Steve Schuman, and Gem Stapleton. Diagrammatic formal specification of a configuration control platform. *ENTCS*, 259:87–104, 2009.

[KAK09] Jörg Kienzle, Wisam Al Abed, and Jacques Klein. Aspect-oriented multi-view modelling. In *AOSD'09*. IEEE, 2009.

[LS87] Jill H. Larkin and Herbert A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.

[LSE05] Marc Lohmann, Stefan Sauer, and Gregor Engels. Executable visual contracts. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 63–70, 2005.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.

[Moo09] Daniel L. Moody. The "physics" of notations: Toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering*, 6(35):756–779, 2009.

[RGF+06] R. Reddy, S. Ghosh, Robert France, G. Straw, J. .M Bieman, E. Song, and G. Gerog. Directives for composing aspect-oriented design class models. *TAOSD LNCS*, 3880:75–105, 2006.

[SCFY96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *Computer*, 29(2), 1996.

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

[SS86] M. Sumner and J. Sitek. Are structured methods for system analysis and design being used. *Journal of Systems Management*, 37(6):18–23, 1986.

[THHS99] Peri Tarr, Harold Hosher, William Harrison, and Stanley M. Sutton. N degrees if seoaration: multi-dimensional separation of concerns. In *ICSE'99*. IEEE, 1999.

[WA04] J. Whittle and J. Araújo. Scenario modelling with aspects. *IEE Proc. Softw.*, 151(4):157–171, 2004.

[WD96] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. PH, 1996.

Appendix A

VCL Model of Secure Simple Bank

This chapter presents the VCL model of secure simple bank, structured around a collection of VCL packages. VCL packages that make the VCL model of secure simple bank are as follows:

- **Package** `Bank` **(section A.1)** encapsulates the problem domain of banking.
- **Package** `Users` **(section A.2)** encapsulates system user-related functionality.
- **Package** `Authentication` **(section A.3)** encapsulates the core of the *authentication* cconcern.
- **Package** `AuthenticationOps` **(section A.4)** extends the `Authentication` package with authentication operations, such as *login* and *logout*.
- **Package** `AccessControl` **(section A.5)** encapsulates the *access control* concern.
- **Package** `Authorisation` **(section A.6)** puts together the packages `Authentication` and `AccessControl` to provide authenticated access-control.
- **Package** `RolesAndTaskBank` **(section A.7)** defines certain sets that are related with *access control* to be used in the customisation of package `Authorisation` for the purpose of secure simple bank.
- **Package** `SecForBank` **(section A.8)** customises generic package `Authorisation` for the purpose of secure simple bank.
- **Package** `BankACJI` **(section A.9)** defines a surrounding interface for package `Bank` to enable the composition of this package with package `AccessControl`.
- **Package** `BankWithJI` **(section A.10)** adds the surrounding interface for access control (package `BankACJI`) to the package `Bank`.
- **Package** `SecBank` **(section A.11)** encapsulates the overall secure simple Bank system, with all the required concerns as defined by the requirements.

A.1 Package `Bank`

This chapter defines package `Bank`, which localises the problem domain of concern of the secure simple Bank system. The Z that is generated for this package is given in appendix B (section B.2).

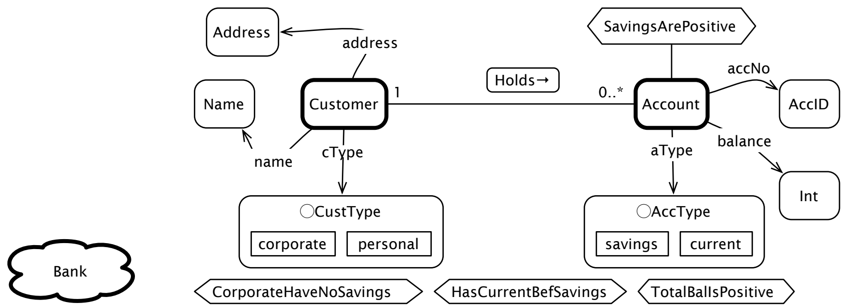


Figure A.1: VCL Package (left) and structural (right) diagrams of package `Bank` (left).

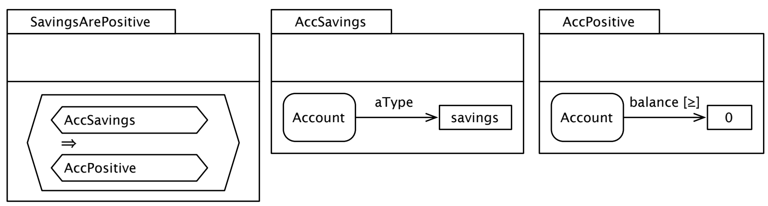


Figure A.2: Assertion diagrams for `Account` invariant `SavingsArePositive`.

A.1.1 Package Definition and Structure

Figure A.1 presents VCL SD of simple Bank. It is as follows:

- Domain blobs `Customer` and `Account` represent main problem domain concepts (requirement $R1$). Property edges `name`, `cType` and `address` define properties of `Customer` (Requirement $R2$); `accNo`, `balance` and `aType` define properties of `Account` (Requirement $R3$).

- Blobs `CustType` and `AccType` are defined by enumeration. `CustType` has elements `corporate` and `personal`; `AccType` has elements `savings` and `current`.

- Relational edge `Holds` relates customers and their accounts. UML-style multiplicity constraints say that a customer may have many accounts and that an account is held by one `Customer` (Requirement $R1$).

- Several invariants constrain state of the system. `SavingsArePositive` is local. Remaining constraints are global: `CorporateHaveNoSavings` (Requirement $R6$), `HasCurrentBeforeSavings` (Requirement $R7$) and `TotalBalanceIsPositive`(Requirement $R5$).

Local Invariants of blob `Account`

This local invariant is described in VCL in Fig. A.2 using three CntDs. All declarations compartments are empty because no extra declarations of names are required to describe the constraint. The first CntD (`SavingsArePositive`) uses VCL's constraint reference expression operators to say that `AccSavings` implies `AccPositive`. Remaining CntDs define constraints

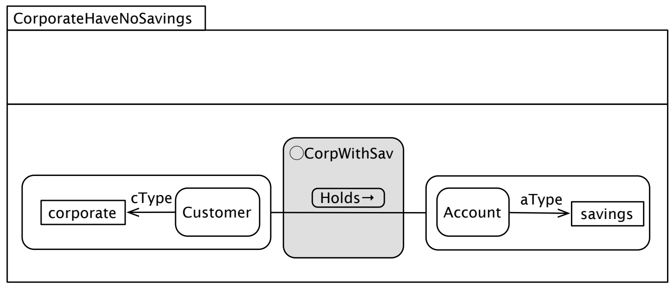


Figure A.3: Assertion diagram for invariant `CorporateHaveNoSavings`.

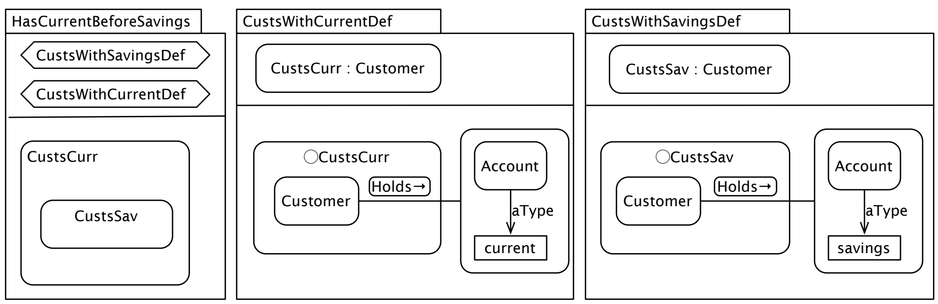


Figure A.4: Assertion diagram for invariant `HasCurrentBeforeSavings`.

`AccSavings` and `AccPositive`. `AccSavings` says that property `aType` of `Account` must be equal to `savings`. `AccPositive` says that property `balance` must be greater or equal to 0.[1]

Global Invariants

Constraint `CorporateHaveNoSavings`. This invariant is described in Fig. A.3. Blob on the left restricts `Customer` to those objects whose property `cType` has value `corporate`. Blob on the right restricts `Account` to those objects whose property `aType` has value `savings`. Outermost blob restrict relation `Holds` to those tuples with corporate customers and savings accounts. Finally, shading is used to say that outermost blob must be empty, which gives required meaning.

Constraint `HasCurrentBeforeSavings`. This invariant is described in Fig. A.4. It defines set of customers with current accounts (`CustCurr`), set of customers with savings accounts (`CustSav`) and then says that latter is subset of former.

Constraint importing results in importing of names. When an imported name is not declared in declarations of importer CntD, then the name is hidden. In `HasCurrentBeforeSavings` `CustSav` and `CustCurr` are not declared, so they are hidden. Names `CustCurr` and `CustSav` refer to same object in the different diagrams. Note the use of *projection* or *extraction* to define domain of a restricted set of tuples of relation `Holds`.

[1] In CntDs, property edges link some blob or object to some expression; by default they denote equality, unless other relational operator is explicitly provided. Above, `aType` edge denotes equality, but `balance` denotes $\geq$.

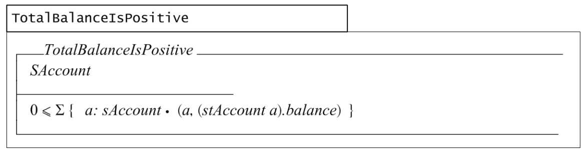


Figure A.5: Z invariant `TotalBalanceIsPositive` embedded in VCL assertion diagram.

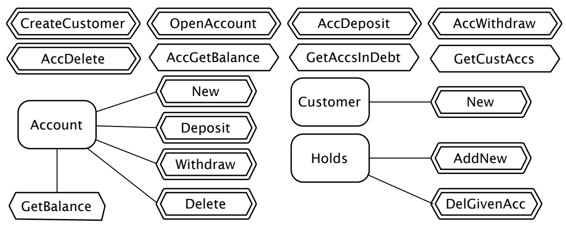


Figure A.6: VCL Behavioural diagram of Simple Bank.

Constraint `TotalBalanceIsPositive`. Not all constraints can be expressed visually in VCL. `TotalBalanceIsPositive` of Bank is such a constraint. VCL gives the specifier the choice of writing a constraint visually or textually. Given a sum operator defined in the target language toolkit (see section C for details), constraint `TotalBalanceIsPositive` is expressed in Z and its text is embedded in a VCL CntD as described in Fig. A.5.

A.1.2 Behavioural Diagram

Behavioural diagram of package `Bank` is given in Fig. A.6. It identify eight system operations, represented as global units, one operation of blob `Customer`, and five operations of blob `Account` and two operations of relational edge `Holds`. These are defined below.

A.1.3 Blob `Customer`.

Figure A.7 presents VCL contract diagram of operation *New* of blob Customer.

A.1.4 Blob `Account`.

Contracts for the local operations of `Account` are as follows:

Contract `New` (Fig. A.8, left). It declares: inputs new account's number (`iAccNo?`) and type (`iAType?`), and output for created object (`a!`). Pre-condition is *true* as its compartment is empty. *Post-condition* says `a!` is given values to its properties; `a!` is to be created: it is on the right but not on the left.

Contract `Delete` (Fig. A.8, centre). This contract declares object to delete as input (`a?`). Pre-condition says that `a?` must have a balance of 0. Post-condition compartment is empty; `a` is to be deleted as is on the left but not on the right.

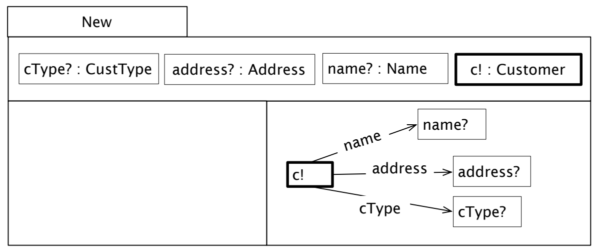


Figure A.7: VCL Contract diagram of `Customer` operation `New`.

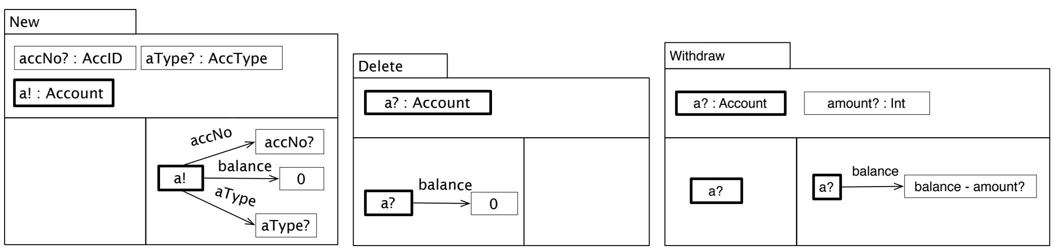


Figure A.8: Contract diagrams of local operations `New`, `Delete` `Withdraw` of `Account`.

Contract `Withdraw` (Fig. A.8, right). It declares two inputs: account from where money is withdrawn (`a?`), and amount to withdraw (`amount?`). Pre-condition requires that `a` exists. Post-condition says that `balance` property of `a?` is given value of expression $balance - amount?$ (where `balance` refers to before-state value).

Contract `Deposit` (Fig. A.9, left). It declares two inputs: account to where money is deposited (`a?`), and amount to deposit (`amount?`). Pre-condition requires that `a` exists. Post-condition says that `balance` property of `a?` is given value of expression $balance + amount?$ (where `balance` refers to before-state value).

Contract `GetBalance` (Fig. A.9, right). This contract describes an observe operation and so it has a single action compartment. It declares as input `Account` object to observe (`a?`), and as output an object to hold `balance` value of a (`accBal!`). Action compartment requires that a does exist (it is an `Account` object of system) and says that `accBal!` holds the `balance` value of `a?`.

A.1.5 Relational Edge `Holds`

Figure A.10 presents VCL contract diagrams for local operations of relational-edge `Holds`.

A.1.6 Global Behaviour

Contract `CreateCustomer` (Fig. A.11, left). It declares inputs `name`, `address` and `cType?` and imports contracts `Customer.New` (see Fig. A.7). `Customer.New` creates a new `Customer` object. `name`, `address` and `cType?` are shared inputs.

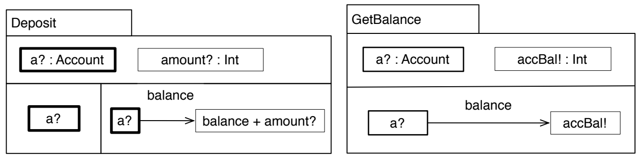


Figure A.9: Contract diagrams of local operations `Deposit` and `GetBalance` of `Account`.

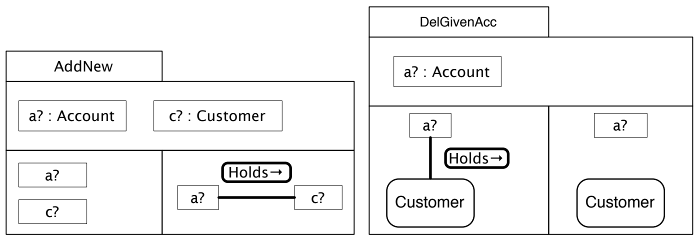


Figure A.10: Contract diagrams of operations `AddNew` and `DelGivenAcc` of relational edge `Holds`.

Contract `OpenAccount` **(Fig. A.11, centre).** It declares inputs `aType?` and `c?`, and imports contracts `Account.New` (see Fig. A.8) and `Holds.AddNew` (see Fig. A.10). `Account.New` creates a new `Account` object. `Holds.AddNew` creates a new tuple of relation `Holds`. `aType?` and `c?` are shared inputs; `accNo?` and `a!` (of `Account.New`) are used internally only. Arrow from blob `AccID` (set of all account identifiers) to `Account.New` means that some object of this set (selected non-deterministically) is passed to `Account.New` through channel `accNo?`. Arrow from contract `Account.New` to `Holds.AddNew` says that output `a!` is to be passed to `AddNew` through input `a?`.

Contract `AccWithdraw` **(Fig. A.11, right).** It imports contract `Account.Withdraw` integrally. Action compartments are empty and so pre- and post-conditions are those of imported contract.

Contract `AccDeposit` **(Fig. A.12, middle).** It imports contract `Account.Deposit` integrally. Action compartments are empty and so pre- and post-conditions are those of imported contract.

Contract `AccDelete` **(Fig. A.12, right).** It imports both `Account.Delete` and `Holds.DelGivenAccount` integrally. Action compartments are empty to mean that pre-and post-condition are conjunction of imported contracts.

Contract `AccGetBalance` **(Fig. A.12, left).** It imports `Account.GetBalance` integrally. Action compartment is empty, meaning that pre-condition and observe expression are those of imported contract.

Contract `GetCustAccs` **(Fig. A.13, left).** It declares queried customer as input (`c?`) and his accounts as output (set `accs!`). Action compartment includes observe expression, which,

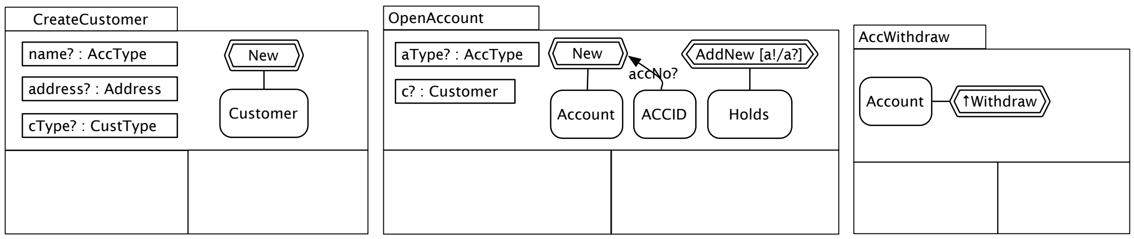


Figure A.11: Contract diagrams of global operations `CreateCustomer`, `OpenAccount` and `AccWithdraw`

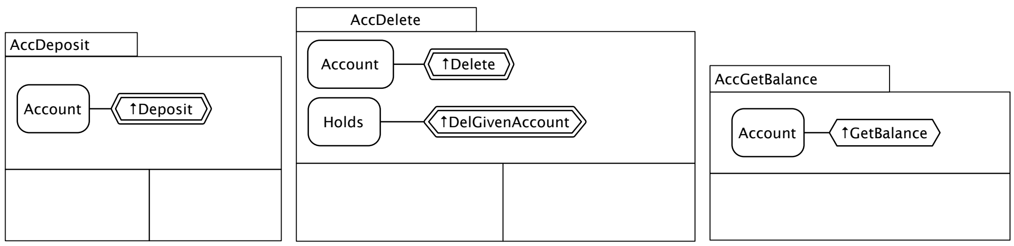


Figure A.12: Contract diagrams of global operations `AccDeposit`, `AccDelete` and `AccGetBalance`

from set of tuples of relation `Holds` restricted to `c`, extracts set of accounts (range of relation) into output `accs!`.

Contract `GetAccsInDebt` (Fig. A.13, right). This observe contract declares output `accs!` to hold set of accounts whose balance is less than 0[2].

A.2 Package `Users`

This chapter defines a package addressing the core of user-related concerns. This is to be extended by other packages defining user-related functionality. The Z that is generated for this package is given in appendix B (section B.3).

A.2.1 Package Definition and Structure

Figure A.14 presents SD of package `Users`. This introduces domain blob `User`, a structure to be used in other packages dealing with user-related concerns. In diagram, value blob `UserStatus` defines a set by enumerating its elements; this says that set `UserStatus` comprises distinct elements named `loggedIn`, `loggedOut` and `blocked`. The labelled arrows emanating from `User` to other blobs define properties possessed by all elements of the set. `User` objects have a user identifier (`uid`), a user or `login` name (`uname`), the actual name of the user (`name`), a password (`pw`) and a record of the number of password misses (`pwMisses`) kept

[2] In contracts, property edges link object to some value; by default they denote equality, unless other relational operator is explicitly provided. In the contracts above, most property edges denote equality; here *balance* denotes $<$.

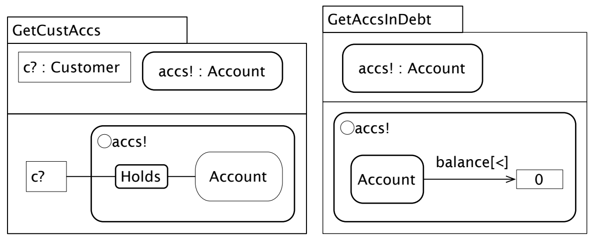


Figure A.13: Contract diagrams of global operations `GetCustAccs` and `GetAccsInDebt`

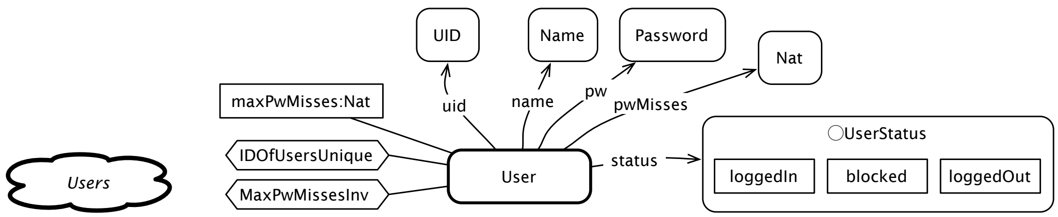


Figure A.14: Package diagram defining package `Users` (left). Structural diagram of `Users` package (right).

for security reasons, and a *login* status (`status`), representing the fact that a user may be *logged-out*, *logged-in* or *blocked* because number of password tries exceeded allowed maximum. Object linked to `User` defines a constant that is visible only in the scope of this blob (a local constant); `maxPwMisses` of blob *Nat* (natural numbers) represents the maximum number of allowed consecutive password misses for some user.

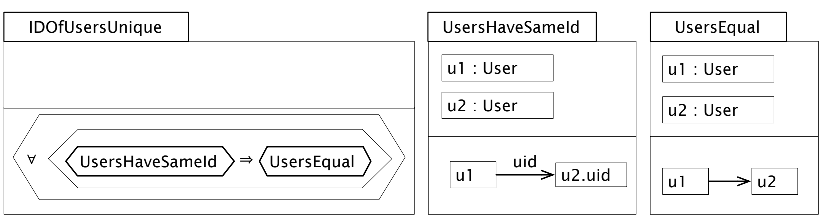


Figure A.15: Local invariant `IDOfUsersUnique` of blob `User`. This says that if two users have the same id, then they must be the same user.

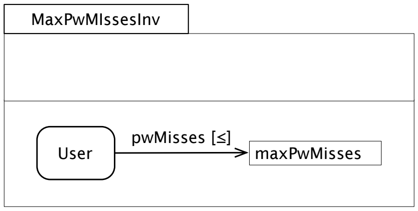


Figure A.16: Local invariant `MaxPwMissesInv` of blob `Users`.

A.2.2 Behaviour

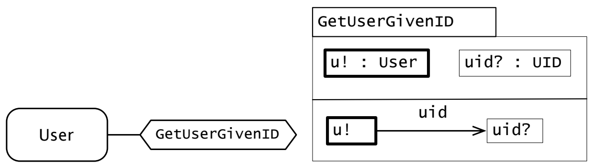


Figure A.17: Behavioural diagram of package `Users` (left). Assertion diagram describing observe operation `GetUserGivenID` of blob `User` (right).

A.3 Package `Authentication`

This chapter defines a package defining the core of a general solution to the concern of user authentication. This is to be extended by other packages to define authentication-related functionality. `Authentication` is package defining a reusable aspect that extends package `Users` (section A.2). The Z that is generated for this package is given in appendix B (section B.4).

A.3.1 Package Definition and Structure

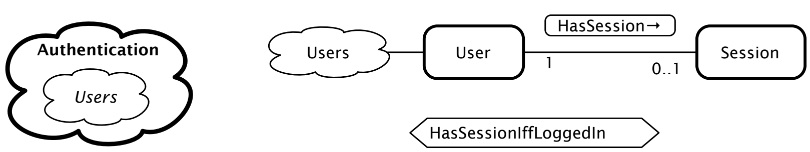


Figure A.18: Package `Authentication` package extends `Users` (left). Global structural diagram of `Authentication` package (right).

Figure A.18 presents the global structural diagram of the `Authentication` package. The diagram is as follows:

- Blob `User` represents a set of users; it comes from package `Users`.
- Blob `Session`, also a domain blob, represents a session that can be opened in the system by some user. Section A.3.1 gives its local definition.

- Relational edge `HasSession` associate a user with its open session. A user can have at most one session open at any one time (multiplicity 0 .. 1), and a session has a user.
- Blob `LoginResult` defines a set with the possible results of a login operation: it may be successful (`loginOk`), the the login may be blocked because maximum limit for password tries has been reached (`isBlocked`), or it may just be that the password is wrong (`wrongPW`).

`Session` Blob

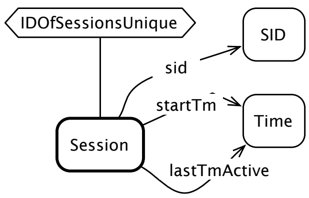


Figure A.19: Local structural diagram of `Session` blob.

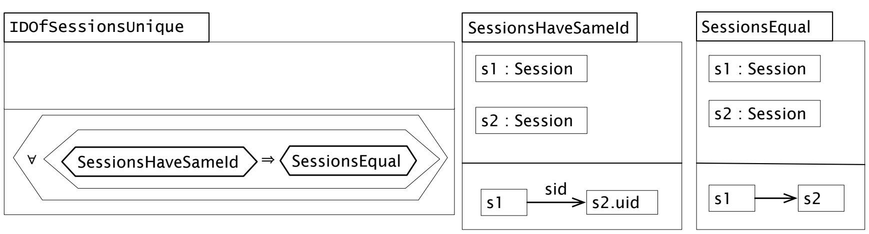


Figure A.20: Constraint diagram for constraint `IdsOfSessionsUnique`.

`User` Blob

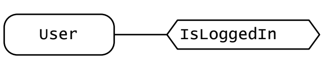


Figure A.21: Local behavioural diagram of `User` blob.

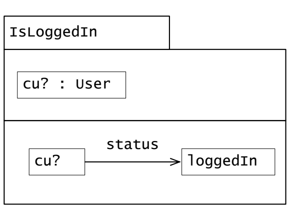


Figure A.22: Assertion diagram for predicate `IsLoggedIn` of Blob `User`.

Global constraints

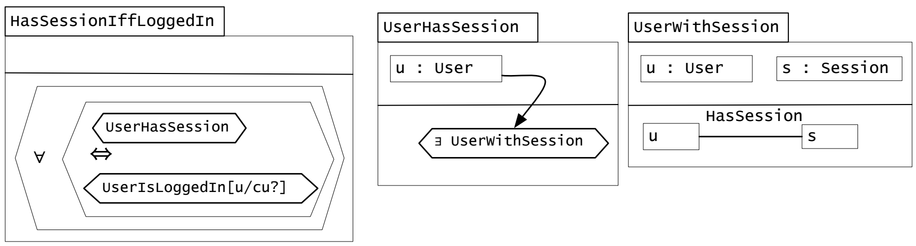


Figure A.23: Constraint diagram for global invariant `HasSessionIfLoggedIn`.

A.3.2 Behaviour

Figure A.24: Behaviour diagram of `Authentication` package.

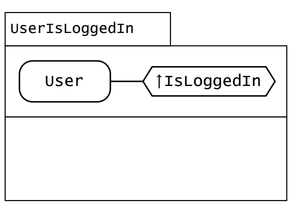


Figure A.25: Assertion diagrams for global predicate `UserIsLoggedIn`.

A.4 Package `AuthenticationOps`

This chapter defines package `AuthenticationOps` that extends package `Authentication` (section A.3) to provide authentication operations, such as login and logout. The Z that is generated for this package is given in appendix B (section B.5).

A.4.1 Package Definition and Structure

Figure A.26: Package `AuthenticationOps` extends package `Authentication` (left). Structural diagram of `AuthenticationOps` package (right).

A.4.2 Behavioural Diagram

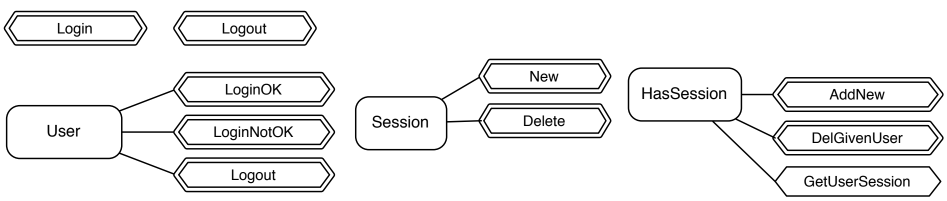


Figure A.27: Behavioural diagram of `AuthenticationOps` package.

A.4.3 Behaviour of Blob `User`

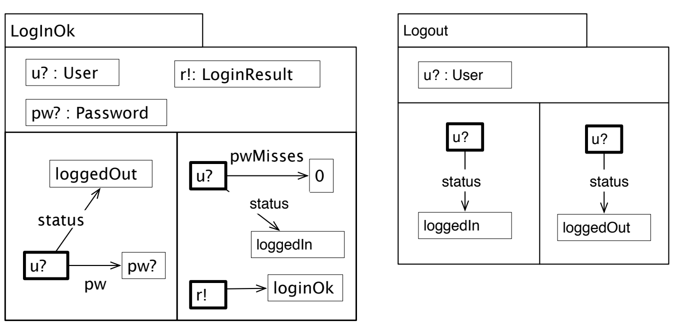


Figure A.28: Contract Diagrams for local operations `LoginOk` and `Logout` of blob `User`.

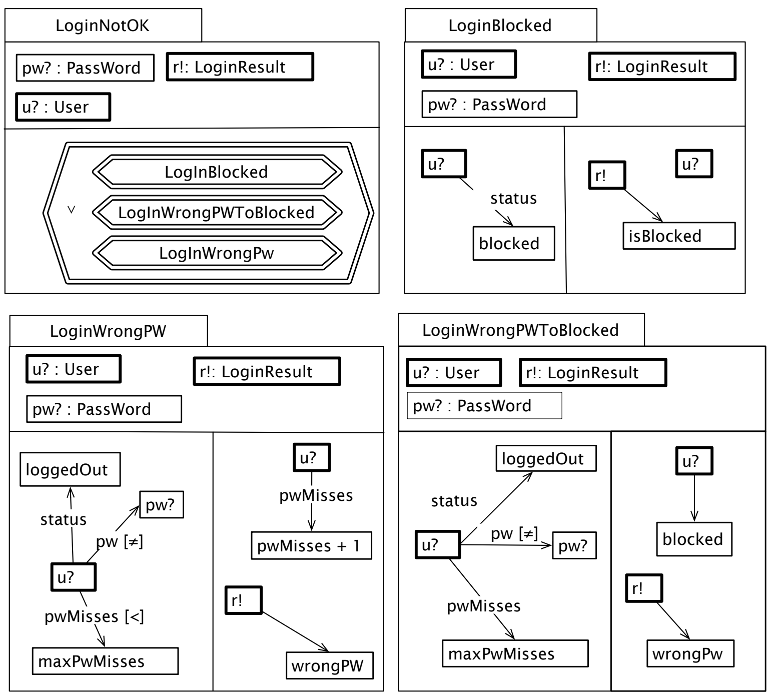


Figure A.29: Contract Diagrams describing local operation `LoginNotOk` of blob `User`.

A.4.4 Behaviour of Blob `Session`

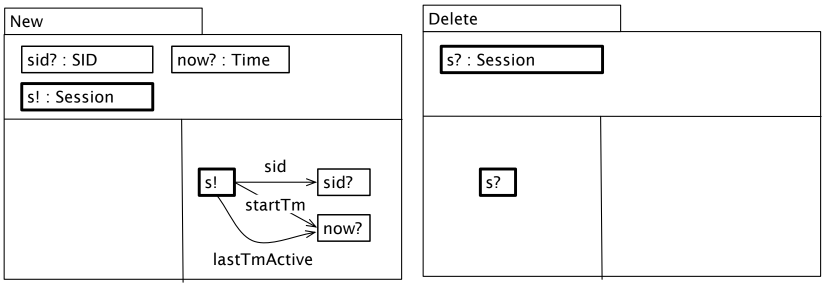


Figure A.30: Contract Diagrams for local operations `New` and `Delete` of blob `Session`.

A.4.5 Behaviour of Blob `HasSession`

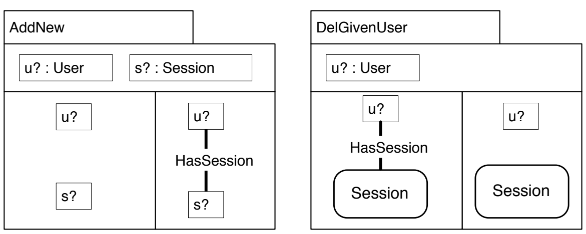


Figure A.31: Contract Diagrams for local operations `AddUserSession` and `DelGivenUser` of relational-edge `HasSession`.

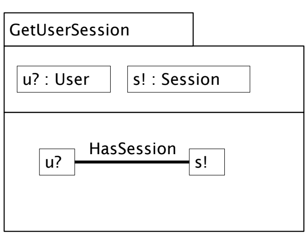


Figure A.32: Assertion diagram describing observe operation `GetUserSession` of relational-edge `HasSession`.

A.4.6 Global Behaviour

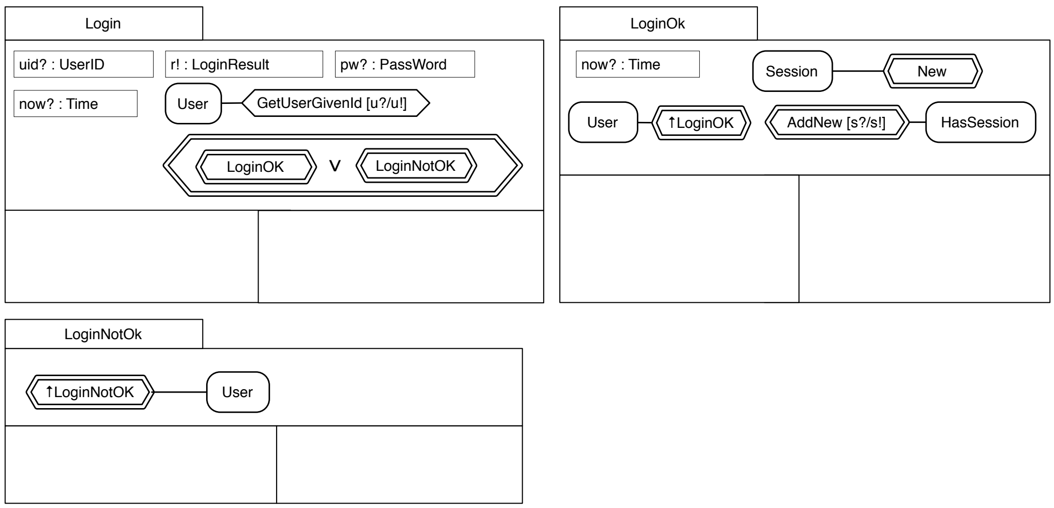


Figure A.33: Contract Diagrams defining global operations `Login`.

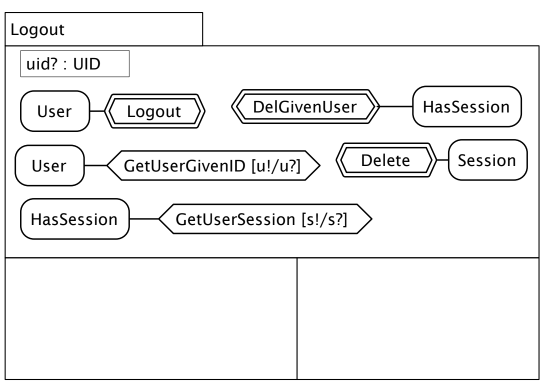


Figure A.34: Contract Diagrams defining global operations `Logout`.

A.5 Package `AccessControl`

This chapter introduces a package representing the core of a solution for rôle-based access control [SCFY96] scheme. Package `AccessControl` extends package `Users` (see chapter A.2). The Z that is generated for this package is given in appendix B (section B.7).

A.5.1 Package Definition and Structure

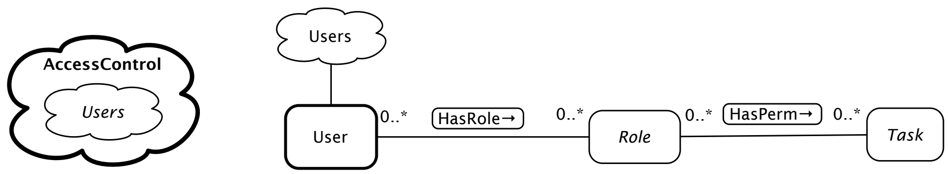


Figure A.35: `AccessControl` package extends `Users` (left). Structural diagram of `AccessControl` package (right).

A.5.2 Behaviour

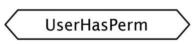


Figure A.36: Behavioural diagram of `AccessControl` package.

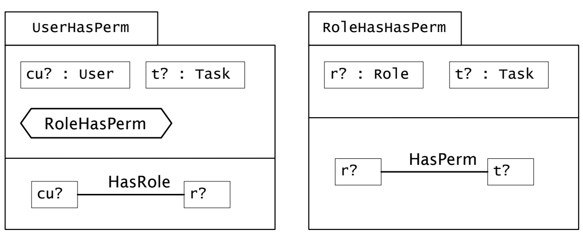


Figure A.37: Assertion diagram defining global observe operation `UserHasPerm`.

A.6 Package `Authorisation`

This chapter introduces package `Authorisation`, which provides both user authentication and rôle-based access control. Package `Authorisation` extends packages `Authentication` (section A.3) and `AccessControl` (section A.5). The Z that is generated for this package is given in appendix B (section B.8).

A.6.1 Structure

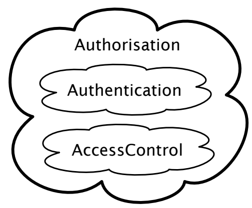


Figure A.38: Package `Authorisation` extends packages `Authentication` and `AccessControl`.

A.6.2 Behaviour

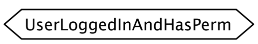


Figure A.39: Behavioural diagram of `Authorisation` package.

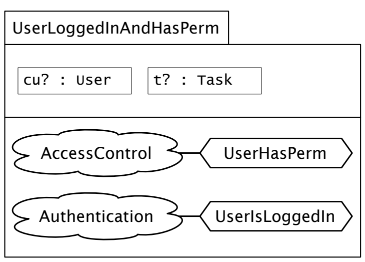


Figure A.40: Assertion diagram describing global observe operation `UserLoggedInAndHasPerm`.

A.7 Package RolesAndTasksBank

This chapter defines package `RolesAndTasksBank`, which defines blobs `Role` and `Task` of generic package `AccessControl` for the needs of secure simple Bank system. It defines those roles and tasks that are appropriate in this context. The Z that is generated for this package is given in appendix B (section B.6).

A.7.1 Package Definition and Structure

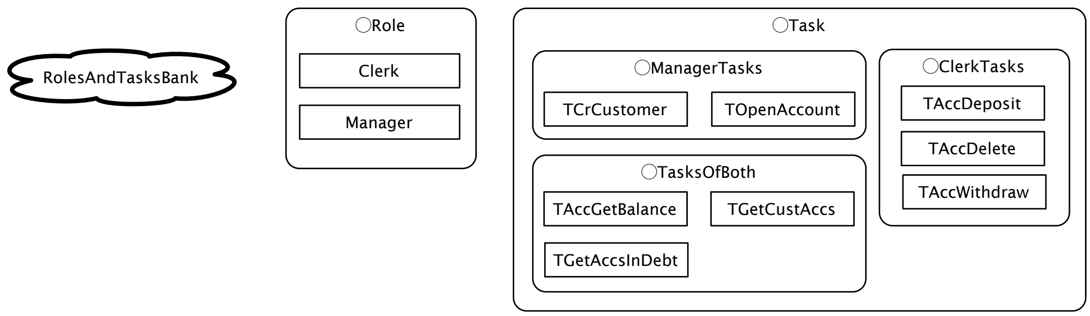


Figure A.41: Package (left) and structural (right) diagrams of package `RolesAndTasksBank`. Structural diagram defines blobs `Role` and `Task` for the purpose of secure simple bank.

A.8 Package SecForBank

This chapter introduces package `SecForBak`, which does the customisation of packages `Authorisation` (section A.6) for the purpose of secure simple bank. The Z that is generated for this package is given in appendix B (section B.9).

A.8.1 Package Definition and Structure

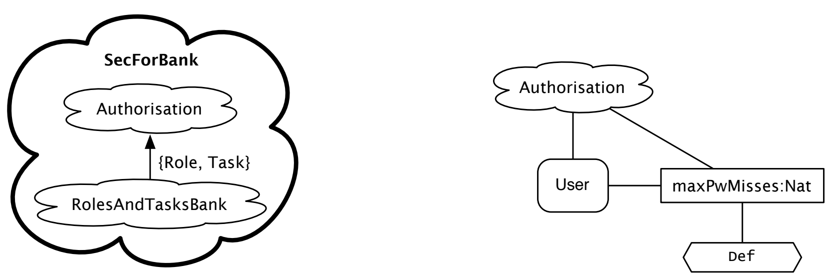


Figure A.42: Package (left) and structural (right) diagrams of package `SecForBank`. `SecForBank` extends `Authorisation`; package `RolesAndTasksBank` overrides blobs defined in `Authorisation`.

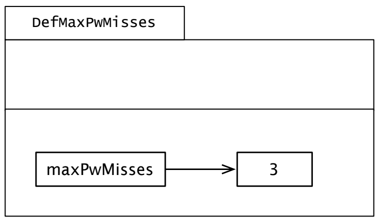


Figure A.43: Assertion diagram defining constant `maxPwMisses` to value 3.

A.8.2 Initialisation

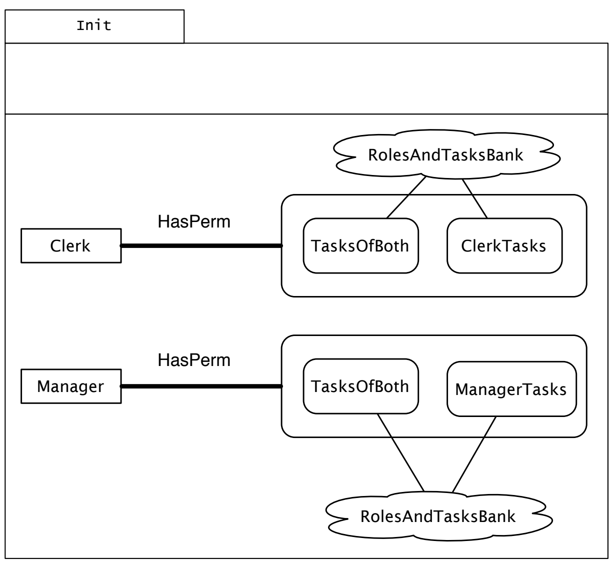


Figure A.44: Initialisation defining initial state of RelationalEdge `HasPerm` of package `AccessControl`.

A.8.3 Behaviour

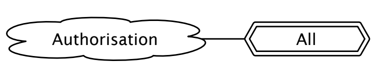


Figure A.45: Behavioural diagram of package `SecForBank`.

A.9 Package BankACJI

This chapter introduces package `BankACJI` (*Bank Access-Control Join Inter- face*), which denes the interface of package `Bank` (section A.1) to the access control concern. The Z that is generated for this package is given in appendix B (section B.10).

A.9.1 Package Definition and Structure

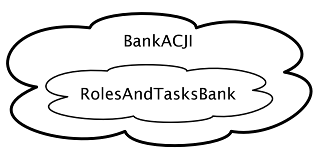


Figure A.46: Package diagram defining package `BankACJI` (left).

A.9.2 Behaviour

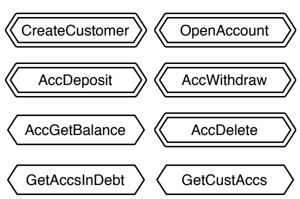


Figure A.47: Behavioural diagram of package `BankACJI`.

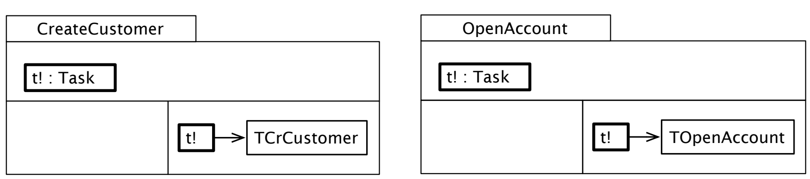


Figure A.48: Contract diagrams for operations `CreateCustomer` and `OpenAccount`.

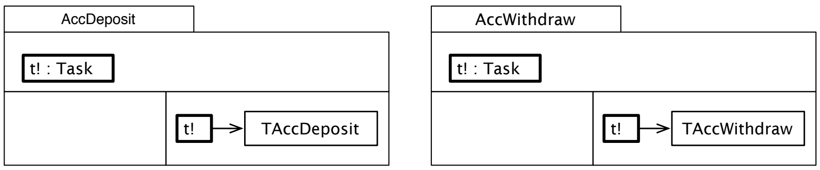


Figure A.49: Contract diagrams for operation `AccDeposit` and `AccWithdraw`.

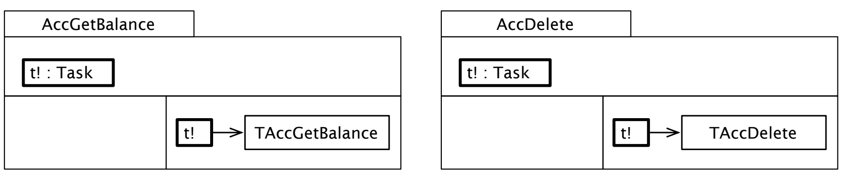


Figure A.50: Contract diagrams for operations `AccGetBalance` and `AccDelete`.

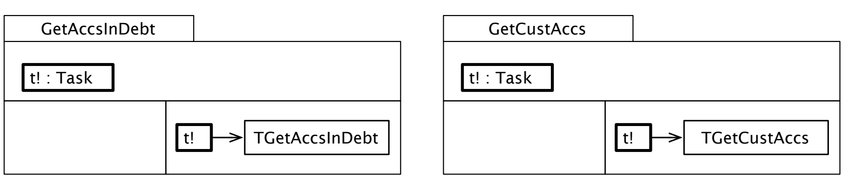


Figure A.51: Contract diagrams for operation `GetAccsInDebt`and `GetCustAccs`.

A.10 Package BankWithJI

This chapter introduces package `BankWithJI` (*Bank With Join Interfaces*), which add the required join interfaces to domain package `Bank` (section A.1). The Z that is generated for this package is given in appendix B (section B.11).

A.10.1 Package Definition and Structure

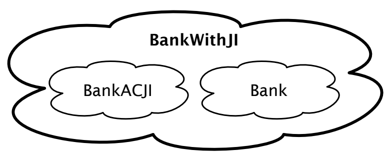


Figure A.52: Package diagram defining package `BankWithJI`.

A.10.2 Behaviour

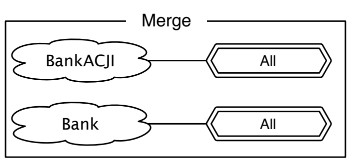


Figure A.53: Behavioural diagram of package `BankWithJI`.

A.11 Package SecBank

This chapter introduces package `SecBank`, which encapsulate the overall secure simple bank system with all its concerns. The Z that is generated for this package is given in appendix B (section B.12).

A.11.1 Package Definition and Structure

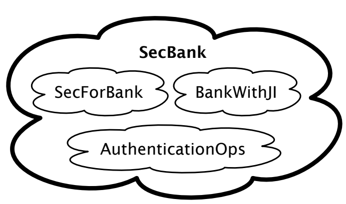


Figure A.54: Package diagram defining package `SecBank`.

A.11.2 Behaviour

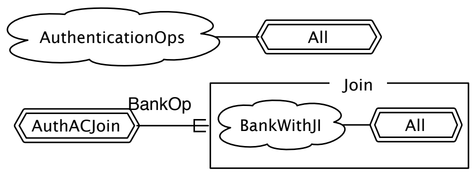


Figure A.55: Behavioural diagram of package `SecBank`.

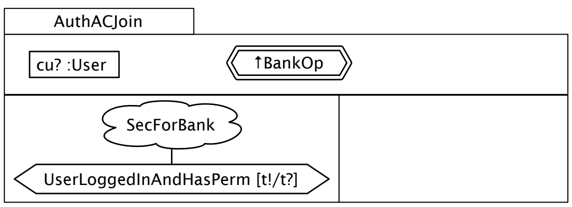


Figure A.56: Contrant diagram for operation `AuthACJoin`.

Appendix B

Z Specification generated from the VCL model of secure simple Bank

This appendix presents the Z specification generated from the VCL model of secure simple Bank (appendix A). Appendix C presents some Z definitions from the ZOO toolkit that are used in the Z specifications presented here. The Z specification presented here has been type-checked using the *Fuzz* Z type-checker.

B.1 Preamble

$$CLASS ::= CustomerCl \mid AccountCl \mid UserCl \mid SessionCl$$

$$\begin{array}{|l}
subCl : CLASS \leftrightarrow CLASS \\
abstractCl : \mathbb{P}\, CLASS \\
rootCl : \mathbb{P}\, CLASS \\
\hline
subCl = \{\} \\
abstractCl = \{\} \\
rootCl = CLASS \setminus \operatorname{dom} subCl
\end{array}$$

$$\begin{array}{|l}
\mathbb{O} : CLASS \rightarrow \mathbb{P}_1\, OBJ \\
\mathbb{O}_x : CLASS \rightarrow \mathbb{P}_1\, OBJ \\
\hline
\text{disjoint}\, \mathbb{O}_x \\
\forall\, cl : CLASS \bullet \\
\quad \mathbb{O}\ cl = \mathbb{O}_x\ cl \cup \bigcup(\mathbb{O}_x(\!|\ (subCl^+)^\sim(\!|\ \{cl\}\ |\!)\ |\!)) \\
\forall\, cl, cl' : CLASS \mid cl \mapsto cl' \in subCl \bullet \\
\quad \mathbb{O}\ cl \subseteq \mathbb{O}\ cl'
\end{array}$$

B.2 Package `Bank`

This section presents Z specification generated for the VCL package `Bank` (section A.1).

B.2.1 Blob `Customer`

Structure

$[NAME, ADDRESS]$

$CUSTTYPE ::= corporate \mid personal$

$NAME \neq \varnothing$

$ADDRESS \neq \varnothing$

$$\begin{array}{|l}\hline \textit{Customer} \\ name : NAME \\ address : ADDRESS \\ cType : CUSTTYPE \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{SCustomer} \\ sCustomer : \mathbb{P}(\mathbb{O}\, CustomerCl) \\ stCustomer : \mathbb{O}\, CustomerCl \leftrightarrow Customer \\ \hline \operatorname{dom} stCustomer = sCustomer \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{SCustomerInit} \\ SCustomer\,' \\ \hline sCustomer' = \varnothing \\ stCustomer' = \varnothing \\ \hline\end{array}$$

Behaviour

$$\begin{array}{|l}\hline \textit{CustomerNew} \\ Customer\,' \\ name? : NAME \\ address? : ADDRESS \\ cType? : CUSTTYPE \\ \hline name' = name? \\ address' = address? \\ cType' = cType? \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \Phi \textit{BankSCustomerN} \\ \Delta SCustomer \\ Customer\,' \\ c! : \mathbb{O}\ CustomerCl \\ \hline c! \in \mathbb{O}_x\ CustomerCl \setminus sCustomer \\ sCustomer' = sCustomer \cup \{c!\} \\ stCustomer' = stCustomer \cup \{(c! \mapsto \theta\, Customer')\} \\ \hline\end{array}$$

$$SCustomerNew == \exists\, Customer\,' \bullet \Phi BankSCustomerN \land CustomerNew$$

B.2.2 Blob `Account`

Structure

$$[ACCID]$$

$$ACCID \neq \varnothing$$

$$AccType ::= current \mid savings$$

Account0

$$
\begin{array}{|l}
accNo : ACCID \\
aType : AccType \\
balance : \mathbb{Z}
\end{array}
$$

AccSavings

$$
\begin{array}{|l}
Account0 \\
\hline
aType = savings
\end{array}
$$

AccPositive

$$
\begin{array}{|l}
Account0 \\
\hline
balance \geq 0
\end{array}
$$

SavingsArePositive

$$
\begin{array}{|l}
Account0 \\
\hline
AccSavings \Rightarrow AccPositive
\end{array}
$$

Account

$$
\begin{array}{|l}
Account0 \\
\hline
SavingsArePositive
\end{array}
$$

SAccount

$$
\begin{array}{|l}
sAccount : \mathbb{P}(\mathbb{O}\ AccountCl) \\
stAccount : \mathbb{O}\ AccountCl \nrightarrow Account \\
\hline
\mathrm{dom}\, stAccount = sAccount
\end{array}
$$

SAccountInit

$$
\begin{array}{|l}
SAccount' \\
\hline
sAccount' = \varnothing \\
stAccount' = \varnothing
\end{array}
$$

Behaviour

AccountNew

$$
\begin{array}{|l}
Account' \\
accNo? : ACCID \\
aType? : AccType \\
\hline
accNo' = accNo? \\
aType' = aType? \\
balance' = 0
\end{array}
$$

AccountDelete

$$
\begin{array}{|l}
Account \\
\hline
balance = 0
\end{array}
$$

AccountDeposit

$$
\begin{array}{|l}
\Delta Account \\
amount? : \mathbb{N} \\
\hline
accNo' = accNo' \\
aType' = aType' \\
balance' = balance + amount?
\end{array}
$$

AccountWithdraw

$$
\begin{array}{|l}
\Delta Account \\
amount? : \mathbb{N} \\
\hline
accNo' = accNo' \\
aType' = aType' \\
balance' = balance - amount?
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{AccountGetBalance} \\
\Xi \textit{Account} \\
\textit{accBal}! : \mathbb{Z} \\
\hline
\textit{accBal}! = \textit{balance} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\Phi \textit{BankSAccountN} \\
\Delta\ \textit{SAccount} \\
\textit{Account}\ ' \\
a! : \mathbb{O}\ \textit{AccountCl} \\
\hline
a! \in \mathbb{O}_x\ \textit{AccountCl} \setminus \textit{sAccount} \\
\textit{sAccount}' = \textit{sAccount} \cup \{a!\} \\
\textit{stAccount}' = \textit{stAccount} \cup \{(a! \mapsto \theta\ \textit{Account}')\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\Phi \textit{BankSAccountU} \\
\Delta\ \textit{SAccount} \\
\Delta\ \textit{Account} \\
a? : \mathbb{O}\ \textit{AccountCl} \\
\hline
a? \in \textit{sAccount} \\
\theta\ \textit{Account} = \textit{stAccount}\ a? \\
\textit{sAccount}' = \textit{sAccount} \\
\textit{stAccount}' = \textit{stAccount} \oplus \{(a? \mapsto \theta\ \textit{Account}')\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\Phi \textit{BankSAccountO} \\
\Xi \textit{SAccount} \\
\Xi \textit{Account} \\
a? : \mathbb{O}\ \textit{AccountCl} \\
\hline
a? \in \textit{sAccount} \\
\theta\ \textit{Account} = \textit{stAccount}\ a? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\Phi \textit{BankSAccountD} \\
\Delta \textit{SAccount} \\
\textit{Account} \\
a? : \mathbb{O}\ \textit{AccountCl} \\
\hline
a? \in \textit{sAccount} \\
\theta \textit{Account} = \textit{stAccount}\ a? \\
\textit{sAccount}' = \textit{sAccount} \setminus \{a?\} \\
\textit{stAccount}' = \{a?\} \lhd \textit{stAccount} \\
\hline
\end{array}
$$

$$\textit{SAccountNew} == \exists\, \textit{Account}\ ' \bullet \Phi \textit{BankSAccountN} \wedge \textit{AccountNew}$$

$$\textit{SAccountDelete} == \exists\, \textit{Account} \bullet \Phi \textit{BankSAccountD} \wedge \textit{AccountDelete}$$

$$\textit{SAccountDeposit} == \exists\, \Delta \textit{Account} \bullet \Phi \textit{BankSAccountU} \wedge \textit{AccountDeposit}$$

$$\textit{SAccountWithdraw} == \exists\, \Delta \textit{Account} \bullet \Phi \textit{BankSAccountU} \wedge \textit{AccountWithdraw}$$

$$\textit{SAccountGetBalance} == \exists\, \Delta \textit{Account} \bullet \Phi \textit{BankSAccountO} \wedge \textit{AccountGetBalance}$$

B.2.3 Relational Edge Holds

Structure

$$\begin{array}{|l}\hline \textit{Holds} \\ rHolds : \mathbb{O}\ CustomerCl \leftrightarrow \mathbb{O} AccountCl \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{HoldsInit} \\ Holds' \\ \hline rHolds' = \varnothing \\ \hline\end{array}$$

Behaviour

$$\begin{array}{|l}\hline \textit{HoldsAddNew} \\ \Delta Holds \\ a? : \mathbb{O}\ AccountCl \\ c? : \mathbb{O}\ CustomerCl \\ \hline rHolds' = rHolds \cup \{(a?, c?)\} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{HoldsDelGivenAccount} \\ \Delta Holds \\ a? : \mathbb{O}\ AccountCl \\ \hline rHolds' = rHolds \rhd \{a?\} \\ \hline\end{array}$$

B.2.4 Global State

$$\begin{array}{|l}\hline \textit{BankSt} \\ SAccount;\ SCustomer;\ Holds \\ \hline\end{array}$$

Constraint of Relational Edge

$$\begin{array}{|l}\hline \textit{BankAHoldsGCnt} \\ BankSt \\ \hline \mathrm{mult}(rHolds, sCustomer, sAccount, om, \{\}, \{\}) \\ \hline\end{array}$$

Constraint from Constraint Diagram `CorporateHaveNoSavings`

$$\begin{array}{|l}\hline \textit{BankCorporateHaveNoSavings} \\ BankSt \\ \hline \{oC : sCustomer \mid (stCustomer\ oC).cType = corporate\} \lhd rHolds \rhd \\ \quad \{oA : sAccount \mid (stAccount\ oA).aType = savings\} = \varnothing \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{DeclCustsCurr} \\ CustsCurr : \mathbb{P}(\mathbb{O} CustomerCl) \\ \hline\end{array}$$

Constraint from Constraint Diagram `HasCurrentBeforeSavings`

$$\begin{array}{|l}
\hline
\textit{CustsWithCurrentDef} \\
\textit{BankSt} \\
\textit{DeclCustsCurr} \\
\hline
\textit{CustsCurr} = \mathrm{dom}(\textit{sCustomer} \lhd \textit{rHolds} \rhd \\
\quad \{\textit{oA} : \textit{sAccount} \mid (\textit{stAccount}\ \textit{oA}).\textit{aType} = \textit{current}\}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{DeclCustsSav} \\
\textit{CustsSav} : \mathbb{P}(\mathbb{O}\,\textit{CustomerCl}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{CustsWithSavingsDef} \\
\textit{BankSt} \\
\textit{DeclCustsSav} \\
\hline
\textit{CustsSav} = \mathrm{dom}(\textit{sCustomer} \lhd \textit{rHolds} \rhd \\
\quad \{\textit{oA} : \textit{sAccount} \mid (\textit{stAccount}\ \textit{oA}).\textit{aType} = \textit{current}\}) \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{HasCurrentBeforeSavings0} \\
\textit{BankSt} \\
\textit{CustsWithCurrentDef} \\
\textit{CustsWithSavingsDef} \\
\hline
\textit{CustsSav} \subseteq \textit{CustsCurr} \\
\hline
\end{array}$$

$$\begin{array}{l}
\textit{BankHasCurrentBeforeSavings} == \exists\, \textit{DeclCustsCurr};\ \textit{DeclCustsSav} \bullet \\
\quad \textit{HasCurrentBeforeSavings}0
\end{array}$$

Constraint from Constraint Diagram `TotalBalanceIsPositive`

$$\begin{array}{|l}
\hline
\textit{BankTotalBalanceIsPositive} \\
\textit{BankSt} \\
\hline
0 \leq \Sigma\{\textit{oA} : \textit{sAccount} \bullet (\textit{oA}, (\textit{stAccount}\ \textit{oA}).\textit{balance})\} \\
\hline
\end{array}$$

Full Definition of package `Bank`**'s state**

$$\begin{array}{|l}
\hline
\textit{Bank} \\
\textit{BankSt} \\
\hline
\textit{BankAHoldsGCnt} \\
\textit{BankCorporateHaveNoSavings} \\
\textit{BankTotalBalanceIsPositive} \\
\textit{BankHasCurrentBeforeSavings} \\
\hline
\end{array}$$

$BankInit == Bank' \wedge SCustomerInit \wedge SAccountInit \wedge HoldsInit$

B.2.5 Operations

Operation `CreateCustomer`

$$\Psi BankCreateCustomer == \Delta Bank \wedge \Xi SAccount \wedge \Xi Holds$$
$$BankCreateCustomer == \Psi BankCreateCustomer \wedge SCustomerNew$$

Operation `OpenAccount`

$$\Psi BankOpenAccount == \Delta\ Bank \wedge \Xi\ Customer$$
$$BankOpenAccount0 == [c? : \mathbb{O}\ CustomerCl;\ \Delta Bank \mid c? \in sCustomer]$$
$$BankConnAccountNew == [accNo? : ACCID \mid accNo? \in ACCID]$$
$$BankOpenAccount == \Psi BankOpenAccount \wedge SAccountNew \wedge BankOpenAccount0 \\ \wedge HoldsAddNew[a!/a?] \wedge BankConnAccountNew \setminus (accNo?, a!)$$

Operation `AccDelete`

$$\Psi BankAccDelete == \Delta\ Bank \wedge \Xi\ Customer$$
$$BankAccDelete == \Psi BankAccDelete \wedge SAccountDelete \wedge HoldsDelGivenAccount$$

Operation `AccDeposit`

$$\Psi BankAccDeposit == \Delta\ Bank \wedge \Xi\ Customer \wedge \Xi\ Holds$$
$$BankAccDeposit == \Psi BankAccDeposit \wedge SAccountDeposit$$

Operation `AccWithdraw`

$$\Psi BankAccWithdraw == \Delta Bank \wedge \Xi Customer \wedge \Xi Holds$$
$$BankAccWithdraw == \Psi BankAccWithdraw \wedge SAccountWithdraw$$

Operation `AccGetBalance`

$$BankAccGetBalance == \Xi Bank \wedge SAccountGetBalance$$

Operations `GetAccsInDebt` **and** `GetCustAccounts`

$BankGetAccsInDebt$

$accs! : \mathbb{P}(\mathbb{O}\ AccountCl)$
$\Xi Bank$

$accs! = \{acc : \mathbb{O}\ AccountCl \mid (stAccount\ acc).balance < 0\}$

$$\begin{array}{|l}
\hline
\textit{BankGetCustAccs} \\
c? : \mathbb{O}\ \textit{CustomerCl} \\
\textit{accs}! : \mathbb{P}(\mathbb{O}\ \textit{AccountCl}) \\
\Xi \textit{Bank} \\
\hline
\textit{accs}! = \operatorname{ran}(\{c?\} \lhd \textit{rHolds}) \\
\hline
\end{array}$$

B.3 Package `Users`

The following presents the Z specification that is generated for VCL package `Users` (section A.2).

B.3.1 Blob `User`

$$[\textit{UID}, \textit{Name}, \textit{Password}]$$

This defines constant `maxPWMisses`.

$$\textit{maxPwMisses} : \mathbb{N}$$

$$\textit{UserStatus} ::= \textit{loggedIn} \mid \textit{blocked} \mid \textit{loggedOut}$$

$$\begin{array}{|l}
\hline
\textit{UserDef} \\
\textit{uid} : \textit{UID} \\
\textit{name} : \textit{Name} \\
\textit{pw} : \textit{Password} \\
\textit{pwMisses} : \mathbb{N} \\
\textit{status} : \textit{UserStatus} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{UserMaxPwMissesInv} \\
\textit{UserDef} \\
\hline
\textit{pwMisses} \leq \textit{maxPwMisses} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{User} \\
\textit{UserDef} \\
\hline
\textit{UserMaxPwMissesInv} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{SUserDef} \\
\textit{sUser} : \mathbb{P}(\mathbb{O}\ \textit{UserCl}) \\
\textit{stUser} : \mathbb{O}\ \textit{UserCl} \rightarrowtail\!\!\!\!\!\!+\ \textit{User} \\
\hline
\operatorname{dom} \textit{stUser} = \textit{sUser} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Declu1} \\
u1 : \mathbb{O}\ \textit{UserCl} \\
\hline
\end{array}$$

$$\begin{array}{|l}
\hline
\textit{Declu2} \\
u2 : \mathbb{O}\ \textit{UserCl} \\
\hline
\end{array}$$

$$\begin{array}{|l}\hline \textit{UsersEqual} \\ \textit{Declu}1 \\ \textit{Declu}2 \\ \hline u1 = u2 \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{UsersHaveSameId} \\ \textit{SUserDef} \\ \textit{Declu}1 \\ \textit{Declu}2 \\ \hline (\textit{stUser}~u1).\textit{uid} = (\textit{stUser}~u2).\textit{uid} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{IDOfUsersUnique} \\ \textit{SUserDef} \\ \hline \forall \textit{Declu}1;~\textit{Declu}2 \bullet \textit{UsersHaveSameId} \Rightarrow \textit{UsersEqual} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{SUser} \\ \textit{SUserDef} \\ \hline \textit{IDOfUsersUnique} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{SUserInit} \\ \textit{SUser}~' \\ \hline \textit{sUser}' = \varnothing \\ \textit{stUser}' = \varnothing \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{SUserGetUserGivenID} \\ \textit{SUser} \\ u! : \mathbb{O}~\textit{UserCl} \\ \textit{uid}? : \textit{UID} \\ \hline (\textit{stUser}~u!).\textit{uid} = \textit{uid}? \\ \hline\end{array}$$

B.4 Package `Authentication`

The following presents the Z specification that is generated for VCL package `Authentication` (section A.3).

B.4.1 Blob `Session`

$$[\textit{SID}]$$

$$\textit{Time} == \mathbb{Z}$$

$$\begin{array}{|l}\hline \textit{Session} \\ \textit{sid} : \textit{SID} \\ \textit{startTm} : \textit{Time} \\ \textit{lastTmActive} : \textit{Time} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{SSessionDef} \\ \textit{sSession} : \mathbb{P}(\mathbb{O}~\textit{SessionCl}) \\ \textit{stSession} : \mathbb{O}~\textit{SessionCl} \nrightarrow \textit{Session} \\ \hline \operatorname{dom} \textit{stSession} = \textit{sSession} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Decls}1 \\ s1 : \mathbb{O}~\textit{SessionCl} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Decls}2 \\ s2 : \mathbb{O}~\textit{SessionCl} \\ \hline\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SessionsEqual}} \\
\hline
Decls1 \\
Decls2 \\
\hline
s1 = s2 \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SessionsHaveSameId}} \\
\hline
SSessionDef \\
Decls1 \\
Decls2 \\
\hline
(stSession\ s1).sid = (stSession\ s2).sid \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{IDOfSessionsUnique}} \\
\hline
SSessionDef \\
\hline
\forall\, Decls1;\ Decls2 \bullet SessionsHaveSameId \Rightarrow SessionsEqual \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SSession}} \\
\hline
SSessionDef \\
\hline
IDOfSessionsUnique \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SSesionInit}} \\
\hline
SSession\,' \\
\hline
sSession' = \varnothing \land stSession' = \varnothing \\
\hline
\end{array}$$

B.4.2 Relational Edge `HasSession`

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{HasSession}} \\
\hline
rHasSession : \mathbb{O}\ UserCl \leftrightarrow \mathbb{O}\ SessionCl \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{HasSessionInit}} \\
\hline
HasSession\,' \\
\hline
rHasSession' = \varnothing \\
\hline
\end{array}$$

B.4.3 Blob `User`

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{SUserIsLoggedIn}} \\
\hline
SUser \\
cu? : \mathbb{O}\ UserCl \\
\hline
(stUser\ cu?).status = loggedIn \\
\hline
\end{array}$$

B.4.4 Global State

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthenticationSt}} \\ \hline
SUser \\
SSession \\
HasSession \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Decls}} \\ \hline
s : \mathbb{O}\ SessionCl \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Declu}} \\ \hline
u : \mathbb{O}\ UserCl \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthenticationUserWithSession}} \\ \hline
AuthenticationSt \\
Declu \\
Decls \\ \hline
(u, s) \in rHasSession \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthenticationUserHasSession}} \\ \hline
AuthenticationSt \\
Declu \\ \hline
\exists\ Decls \bullet AuthenticationUserWithSession \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthenticationHasSessionIfLoggedIn}} \\ \hline
AuthenticationSt \\ \hline
\forall\ Declu \bullet AuthenticationUserHasSession \Leftrightarrow SUserIsLoggedIn[u/cu?] \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthenticationHasSessionGCnt}} \\ \hline
AuthenticationSt \\ \hline
\mathrm{mult}(rHasSession, sUser, sSession, ozo, \{\}, \{\}) \\ \hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Authentication}} \\ \hline
AuthenticationSt \\ \hline
AuthenticationHasSessionIfLoggedIn \\
AuthenticationHasSessionGCnt \\ \hline
\end{array}$$

$$AuthenticationInit == Authentication\ ' \land SUserInit \land SSesionInit \land HasSessionInit$$

B.4.5 Global Behaviour

$$AuthenticationUserIsLoggedIn == Authentication \land SUserIsLoggedIn$$

B.5 Package `AuthenticationOps`

The following presents the Z specification that is generated for VCL package `AuthenticationOps` (section A.4).

B.5.1 Blob `User`

$$LoginResult ::= loginOK \mid wrongPW \mid isBlocked$$

$$\begin{array}{l}
\textit{UserLoginOk} \\
\hline
\Delta\ User \\
pw? : Password \\
r! : LoginResult \\
\hline
status = loggedOut \\
pw = pw? \\
pwMisses' = 0 \\
status' = loggedIn \\
pw' = pw \\
name' = name \\
uid' = uid \\
r! = loginOK
\end{array}$$

$$\begin{array}{l}
\textit{UserLogout} \\
\hline
\Delta\ User \\
\hline
status = loggedIn \\
status' = loggedOut \\
pwMisses' = pwMisses \\
pw' = pw \\
name' = name \\
uid' = uid
\end{array}$$

$$\begin{array}{l}
\textit{UserLoginBlocked} \\
\hline
\Delta\ User \\
r! : LoginResult \\
\hline
status = blocked \\
status' = status \\
pwMisses' = pwMisses \\
pw' = pw \land name' = name \\
uid' = uid \\
r! = isBlocked
\end{array}$$

$$\begin{array}{l}
\textit{UserLoginWrongPW} \\
\hline
\Delta\ User \\
pw? : Password \\
r! : LoginResult \\
\hline
status = loggedOut \\
pw \neq pw? \\
pwMisses < maxPwMisses \\
status' = status \\
pwMisses' = pwMisses + 1 \\
pw' = pw \land name' = name \\
uid' = uid \\
r! = wrongPW
\end{array}$$

$$\begin{array}{l}
\textit{UserLoginWrongPWToBlocked} \\
\hline
\Delta\ User \\
pw? : Password \\
r! : LoginResult \\
\hline
status = loggedOut \\
pw \neq pw? \\
pwMisses = maxPwMisses \\
status' = blocked \\
pwMisses' = pwMisses \\
pw' = pw \land name' = name \land uid' = uid \\
r! = wrongPW
\end{array}$$

$\Phi SUserNew$

$$\begin{array}{|l}
\Delta\ SUser \\
User' \\
u! : \mathbb{O}\ UserCl \\
\hline
u! \in \mathbb{O}_x\ UserCl \setminus sUser \\
sUser' = sUser \cup \{u!\} \\
stUser' = stUser \cup \{(u!, \theta\ User')\}
\end{array}$$

$\Phi SUserDel$

$$\begin{array}{|l}
\Delta SUser \\
User \\
u? : \mathbb{O}\ UserCl \\
\hline
u? \in sUser \\
\theta User = stUser\ u? \\
sUser' = sUser \setminus \{u?\} \\
stUser' = \{u?\} \lhd stUser
\end{array}$$

$\Phi SUserUpd$

$$\begin{array}{|l}
\Delta SUser \\
\Delta User \\
u? : \mathbb{O}\ UserCl \\
\hline
u? \in sUser \\
\theta User = stUser\ u? \\
sUser' = sUser \\
stUser' = stUser \oplus \{(u?, \theta User')\}
\end{array}$$

$$UserLoginNotOk == UserLoginBlocked \lor UserLoginWrongPW$$
$$\lor\ UserLoginWrongPWToBlocked$$

$$SUserLoginOk == \exists\, \Delta User \bullet \Phi SUserUpd \land UserLoginOk$$

$$SUserLogout == \exists\, \Delta User \bullet \Phi SUserUpd \land UserLogout$$

$$SUserLoginNotOk == \exists\, \Delta User \bullet \Phi SUserUpd \land UserLoginNotOk$$

B.5.2 Blob `Session`

$SessionNew$

$$\begin{array}{|l}
Session' \\
sid? : SID \\
now? : Time \\
\hline
sid' = sid? \\
startTm' = now? \\
lastTmActive' = now?
\end{array}$$

$SessionDelete$

$$\begin{array}{|l}
Session
\end{array}$$

$\Phi SSessionNew$

$$\begin{array}{|l}
\Delta SSession \\
Session' \\
s! : \mathbb{O}\ SessionCl \\
\hline
s! \in \mathbb{O}_x\ SessionCl \setminus sSession \\
sSession' = sSession \cup \{s!\} \\
stSession' = stSession \cup \{(s! \mapsto \theta Session')\}
\end{array}$$

$$
\begin{array}{|l}
\hline \Phi SSessionUpd \\
\Delta SSession \\
\Delta Session \\
s? : \mathbb{O}\ SessionCl \\
\hline
s? \in sSession \\
\theta Session = stSession\ s? \\
sSession' = sSession \\
stSession' = stSession \oplus \{(s?, \theta Session')\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline \Phi SSessionO \\
SSession \\
Session \\
s? : \mathbb{O}\ SessionCl \\
\hline
s? \in sSession \\
\theta Session = stSession\ s? \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline \Phi SSessionDel \\
\Delta SSession \\
Session \\
s? : \mathbb{O} SessionCl \\
\hline
s? \in sSession \\
\theta Session = stSession\ s? \\
sSession' = sSession \setminus \{s?\} \\
stSession' = \{s?\} \lhd stSession \\
\hline
\end{array}
$$

$SSessionNew == \exists\, Session' \bullet \Phi SSessionNew \land SessionNew$

$SSessionDelete == \exists\, Session \bullet \Phi SSessionDel \land SessionDelete$

B.5.3 Relational Edge `HasSession`

$$
\begin{array}{|l}
\hline HasSessionAddNew \\
\Delta HasSession \\
u? : \mathbb{O}\ UserCl \\
s? : \mathbb{O}\ SessionCl \\
\hline
rHasSession' = rHasSession \cup \{(u?, s?)\} \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline HasSessionDelGivenUser \\
\Delta HasSession \\
u? : \mathbb{O}\ UserCl \\
\hline
rHasSession' = \{u?\} \lhd rHasSession \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline HasSessionGetUserSession \\
HasSession \\
u? : \mathbb{O}\ UserCl \\
s! : \mathbb{O}\ SessionCl \\
\hline
(u?, s!) \in rHasSession \\
\hline
\end{array}
$$

B.5.4 Global State

$$\begin{array}{|l}\hline \multicolumn{1}{l}{\mathit{AuthenticationOps}} \\ \mathit{Authentication} \\ \hline\end{array}$$

$$AuthenticationOpsInit == Authentication' \land AuthenticationInit$$

B.5.5 Global Behaviour

$$\Psi AuthenticationOpsLoginOk == \Delta AuthenticationOps$$

$$AuthenticationOpsLoginOk == \Psi AuthenticationOpsLoginOk \land SUserLoginOk \\ \quad \land SSessionNew \land HasSessionAddNew[s!/s?] \setminus (sid?, s!)$$

$$\Psi AuthenticationOpsLoginNotOk == \\ \quad \Delta AuthenticationOps \land \Xi SSession \land \Xi HasSession$$

$$AuthenticationOpsLoginNotOk == \\ \quad \Psi AuthenticationOpsLoginNotOk \land SUserLoginNotOk$$

$$AuthenticationOpsLogin == SUserGetUserGivenID[u?/u!] \land \\ \quad (AuthenticationOpsLoginOk \lor AuthenticationOpsLoginNotOk) \setminus (u?)$$

$$\Psi AuthenticationOpsLogout == \Delta AuthenticationOps$$

$$AuthenticationOpsLogout == \Psi AuthenticationOpsLogout \\ \quad \land SUserGetUserGivenID[u?/u!] \\ \quad \land SUserLogout \land HasSessionDelGivenUser \\ \quad \land HasSessionGetUserSession[s?/s!] \\ \quad \land SessionDelete \setminus (u?, s?)$$

B.6 Package `RolesAndTasksBank`

The following presents the Z specification that is generated for VCL package `RolesAndTasksBank` (section A.7).

B.6.1 Blob *Role*

$$Role ::= Clerk \mid Manager$$

B.6.2 Blob *Task*

$$Task ::= TCreateCustomer \mid TOpenAccount \mid TAccDeposit \mid TAccWithdraw \\ \quad \mid TAccGetBalance \mid TAccDelete \mid TGetAccsInDebt \mid TGetCustAccs$$

$$\begin{array}{|l} ClerkTasks, ManagerTasks, TasksOfBoth : \mathbb{P}\ Task \\ \hline ClerkTasks = \{TAccDeposit, TAccWithdraw\} \\ ManagerTasks = \{TCreateCustomer, TOpenAccount, TAccDelete\} \\ TasksOfBoth = \{TAccGetBalance, TGetAccsInDebt, TGetCustAccs\} \end{array}$$

B.7 Package `AccessControl`

The following presents the Z specification that is generated for VCL package `AccessControl` (section A.5).

B.7.1 RelationalEdge `HasRole`

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HasRole}} \\
\hline
rHasRole : \mathbb{O}\ UserCl \leftrightarrow Role \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HasRoleInit}} \\
\hline
HasRole\ ' \\
\hline
rHasRole' = \varnothing \\
\hline
\end{array}
$$

B.7.2 RelationalEdge `HasPerm`

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HasPerm}} \\
\hline
rHasPerm : Role \leftrightarrow Task \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{HasPermInit}} \\
\hline
HasPerm\ ' \\
\hline
rHasPerm' = \varnothing \\
\hline
\end{array}
$$

B.7.3 Global State

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccessControlSt}} \\
\hline
SUser \\
HasRole \\
HasPerm \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccessControlHasRoleGCnt}} \\
\hline
AccessControlSt \\
\hline
\mathrm{mult}(rHasRole, sUser, Role, mm, \{\}, \{\}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccessControlHasPermGCnt}} \\
\hline
AccessControlSt \\
\hline
\mathrm{mult}(rHasPerm, Role, Task, mm, \{\}, \{\}) \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccessControl}} \\
\hline
AccessControlSt \\
\hline
AccessControlHasRoleGCnt \\
AccessControlHasPermGCnt \\
\hline
\end{array}
$$

$$AccessControlInit == AccessControl' \land SUserInit \land HasRoleInit \land HasPermInit$$

B.7.4 Global Behaviour

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{rDeclRoleHasPerm}} \\
r? : Role \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{tDeclRoleHasPerm}} \\
t? : Task \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{RoleHasPerm}} \\
AccessControl \\
rDeclRoleHasPerm \\
tDeclRoleHasPerm \\
\hline
(r?, t?) \in rHasPerm \\
\hline
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AccessControlUserHasPerm0}} \\
AccessControl \\
RoleHasPerm \\
cu? : \mathbb{O}\ UserCl \\
t? : Task \\
\hline
(cu?, r?) \in rHasRole \\
\hline
\end{array}$$

$$AccessControlUserHasPerm == \ \exists\ rDeclRoleHasPerm \bullet AccessControlUserHasPerm0$$

B.8 Package `Authorisation`

The following presents the Z specification that is generated for VCL package `Authorisation` (section A.6).

B.8.1 Global State

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{Authorisation}} \\
Authentication \\
AccessControl \\
\hline
\end{array}$$

B.8.2 Global Behaviour

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{AuthorisationUserLoggedInAndHasPerm}} \\
Authorisation \\
AccessControlUserHasPerm \\
AuthenticationUserIsLoggedIn \\
\hline
\end{array}$$

B.9 Package `SecForBank`

The following presents the Z specification that is generated for VCL package `SecForBank` (section A.8).

B.9.1 Global State

This gives a value to constant `maxPwMisses` of package `Users`:

$$maxPwMisses = 3$$

$$\begin{array}{|l}\hline \textit{SecForBank} \\ \textit{Authorisation} \\ \hline\end{array}$$

$$AccessControlInitMod == AccessControlInit \setminus (rHasPerm')$$

$$\begin{array}{|l}\hline \textit{SecForBankInit} \\ SecForBank\ ' \\ AuthenticationInit \\ AccessControlInitMod \\ \hline rHasPerm\ ' = (\{Clerk\} \times (ClerkTasks \cup TasksOfBoth)) \\ \quad \cup(\{Manager\} \times (ManagerTasks \cup TasksOfBoth)) \\ \hline\end{array}$$

B.9.2 Global Behaviour

$$SecForBankUserLoggedInAndHasPerm == SecForBank \\ \quad \land\ AuthorisationUserLoggedInAndHasPerm$$

B.10 Package `BankACJI`

The following presents the Z specification that is generated for VCL package `BankACJI` (section A.9).

B.10.1 Global Behaviour

$$\begin{array}{|l}\hline \textit{BankACJICreateCustomer} \\ t! : Task \\ \hline t! = TCreateCustomer \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{BankACJIOpenAccount} \\ t! : Task \\ \hline t! = TOpenAccount \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{BankACJIAccDeposit} \\ t! : Task \\ \hline t! = TAccDeposit \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{BankACJIAccWithdraw} \\ t! : Task \\ \hline t! = TAccWithdraw \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{BankACJIAccGetBalance} \\ t! : Task \\ \hline t! = TAccGetBalance \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{BankACJIAccDelete} \\ t! : Task \\ \hline t! = TAccDelete \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{BankACJIGetAccsInDebt} \\ t! : Task \\ \hline t! = TGetAccsInDebt \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{BankACJIGetCustAccs} \\ t! : Task \\ \hline t! = TGetCustAccs \\ \hline\end{array}$$

B.11 Package `BankWithJI`

The following presents the Z specification that is generated for VCL package `BankWithJI` (section A.10).

B.11.1 Global State

BankWithJI
Bank

$$BankWithJIInit == BankWithJI' \land BankInit$$

B.11.2 Global Behaviour

Defines the frame for *update* operations.

$$CommonWithBank == BankWithJI \upharpoonright Bank$$

$$BankWithJIWithoutBank == \exists\, CommonWithBank \bullet BankWithJI$$

$$\Psi BankWithJIMergeOps == \Delta BankWithJI \land \Xi BankWithJIWithoutBank$$

BankWithJICreateCustomer
$\Psi BankWithJIMergeOps$
BankCreateCustomer
BankACJICreateCustomer

BankWithJIOpenAccount
$\Psi BankWithJIMergeOps$
BankOpenAccount
BankACJIOpenAccount

BankWithJIAccDeposit
$\Psi BankWithJIMergeOps$
BankAccDeposit
BankACJIAccDeposit

BankWithJIAccWithdraw
BankAccWithdraw
BankACJIAccWithdraw

BankWithJIAccGetBalance
BankWithJI
BankAccGetBalance
BankACJIAccGetBalance

BankWithJIAccDelete
$\Psi BankWithJIMergeOps$
BankAccDelete
BankACJIAccDelete

BankWithJIGetAccsInDebt
BankWithJI
BankGetAccsInDebt
BankACJIGetAccsInDebt

BankWithJIGetCustAccs
BankWithJI
BankGetCustAccs
BankACJIGetCustAccs

B.12 Package `SecBank`

The following presents the Z specification that is generated for VCL package `SecBank` (section A.11).

B.12.1 Global State

$$\begin{array}{|l}\hline \textit{SecBank} \\ \textit{SecForBank} \\ \textit{BankWithJI} \\ \textit{AuthenticationOps} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{SecBankInit} \\ \textit{SecForBankInit} \\ \textit{BankWithJIInit} \\ \textit{AuthenticationOpsInit} \\ \hline\end{array}$$

B.12.2 Global Bebhaviour

$$\textit{CommonWithAuthenticationOps} == \textit{SecBank} \upharpoonright \textit{AuthenticationOps}$$
$$\textit{SecBankWithoutAuthenticationOps} == \exists\, \textit{CommonWithAuthenticationOps} \bullet \textit{SecBank}$$
$$\Psi \textit{SecBankOpsFromAuthenticationOps} == \Delta \textit{SecBank} \wedge \Xi \textit{SecBankWithoutAuthenticationOps}$$

$$\begin{array}{|l}\hline \textit{Login} \\ \Psi \textit{SecBankOpsFromAuthenticationOps} \\ \textit{AuthenticationOpsLogin} \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{Logout} \\ \Psi \textit{SecBankOpsFromAuthenticationOps} \\ \textit{AuthenticationOpsLogout} \\ \hline\end{array}$$

$$\textit{CommonWithBankWithJI} == \textit{SecBank} \upharpoonright \textit{BankWithJI}$$
$$\textit{SecBankWithoutBankWithJi} == \exists\, \textit{CommonWithBankWithJI} \bullet \textit{SecBank}$$
$$\Psi \textit{SecBankOpsFromBankWithJI} == \Delta \textit{SecBank} \wedge \Xi \textit{SecBankWithoutBankWithJi}$$

$$\begin{array}{|l}\hline \textit{CreateCustomer} \\ \Psi \textit{SecBankOpsFromBankWithJI} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJICreateCustomer} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{OpenAccount} \\ \Psi \textit{SecBankOpsFromBankWithJI} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJIOpenAccount} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline\end{array}$$

$$\begin{array}{|l}\hline \textit{AccDeposit} \\ \Psi \textit{SecBankOpsFromBankWithJI} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJIAccDeposit} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{AccWithdraw} \\ \Psi \textit{SecBankOpsFromBankWithJI} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJIAccWithdraw} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{AccGetBalance} \\ \textit{SecBank} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJIAccGetBalance} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{AccDelete} \\ \Psi \textit{SecBankOpsFromBankWithJI} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJIAccDelete} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{AccGetAccsInDebt} \\ \textit{SecBank} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJIAccDeposit} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline \end{array}$$

$$\begin{array}{|l}\hline \textit{AccGetCustAccs} \\ \textit{SecBank} \\ cu? : \mathbb{O}\,\textit{UserCl} \\ \textit{BankWithJIAccWithdraw} \\ \hline \textit{SecForBankUserLoggedInAndHasPerm}[t!/t?] \\ \hline \end{array}$$

Appendix C

ZOO Toolkit

This appendix presents the generics toolkit of the ZOO style (taken from [Amá07]).

$[OBJ]$

$$
\begin{array}{l}
[L] \\
\hline
\Sigma : (L \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{Z}) \rightarrow \mathbb{Z} \\
\hline
\Sigma \; \{\} = 0 \\
\forall\, l : L;\ n : \mathbb{Z} \bullet \Sigma \; \{(l, n)\} = n \\
\forall\, l : L;\ n : \mathbb{Z};\ S : L \mathrel{+\!\!\!\twoheadrightarrow} \mathbb{Z} \mid \neg\, l \in \operatorname{dom} \ S \bullet \Sigma \; (\{(l, n)\} \cup S) = n + \Sigma \ S \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\mathit{MultTy} ::= \mathit{mm} \mid \mathit{mo} \mid \mathit{om} \mid \mathit{mzo} \mid \mathit{zom} \mid \mathit{oo} \mid \mathit{zozo} \mid \mathit{zoo} \mid \mathit{ozo} \mid \mathit{ms} \mid \mathit{sm} \mid \mathit{ss} \\
\quad \mid \mathit{so} \mid \mathit{os} \mid \mathit{szo} \mid \mathit{zos}
\end{array}
$$

$$
\begin{array}{l}
[X, Y] \\
\hline
\mathrm{mult}\,\_ : \mathbb{P}((X \leftrightarrow Y) \times \mathbb{P}\,X \times \mathbb{P}\,Y \times \mathit{MultTy} \times \mathbb{F}\,\mathbb{N} \times \mathbb{F}\,\mathbb{N}) \\
\hline
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{mm}, s_1, s_2)) \Leftrightarrow r \in \mathit{sx} \leftrightarrow \mathit{sy} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{mo}, s_1, s_2)) \Leftrightarrow r \in \mathit{sx} \rightarrow \mathit{sy} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{om}, s_1, s_2)) \Leftrightarrow r^{\sim} \in \mathit{sy} \rightarrow \mathit{sx} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{mzo}, s_1, s_2)) \Leftrightarrow r \in \mathit{sx} \nrightarrow \mathit{sy} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{zom}, s_1, s_2)) \Leftrightarrow r^{\sim} \in \mathit{sy} \nrightarrow \mathit{sx} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{oo}, s_1, s_2)) \Leftrightarrow r \in \mathit{sx} \rightarrowtail\!\!\!\!\twoheadrightarrow \mathit{sy} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{zozo}, s_1, s_2)) \Leftrightarrow r \in \mathit{sx} \rightarrowtail\!\!\!\!\nrightarrow \mathit{sy} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{zoo}, s_1, s_2)) \Leftrightarrow r \in \mathit{sx} \rightarrowtail \mathit{sy} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{ozo}, s_1, s_2)) \Leftrightarrow r^{\sim} \in \mathit{sy} \rightarrowtail \mathit{sx} \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{ms}, s_1, s_2)) \Leftrightarrow (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{mm}, s_1, s_2)) \\
\qquad\qquad \land (\forall\, x : \mathrm{dom}\, r \bullet \#(\{x\} \lhd r) \in s_1) \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{sm}, s_1, s_2)) \Leftrightarrow (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{mm}, s_1, s_2)) \\
\qquad\qquad \land (\forall\, y : \mathrm{ran}\, r \bullet \#(r \rhd \{y\}) \in s_1) \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{ss}, s_1, s_2)) \Leftrightarrow (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{ms}, s_1, \{\})) \\
\qquad\qquad \land (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{sm}, s_2, \{\})) \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{so}, s_1, s_2)) \Leftrightarrow (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{mo}, s_1, s_2)) \\
\qquad\qquad \land (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{sm}, s_1, s_2)) \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{os}, s_1, s_2)) \Leftrightarrow (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{om}, \{\}, \{\})) \\
\land (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{ms}, s_1, \{\})) \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{szo}, s_1, s_2)) \Leftrightarrow (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{mzo}, \{\}, \{\})) \\
\qquad\qquad \land (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{sm}, s_1, \{\})) \\
\forall\, r : X \leftrightarrow Y;\ \mathit{sx} : \mathbb{P}\,X;\ \mathit{sy} : \mathbb{P}\,Y;\ s_1, s_2 : \mathbb{F}\,\mathbb{N} \bullet \\
\quad (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{zos}, s_1, s_2)) \Leftrightarrow (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{zom}, \{\}, \{\})) \\
\qquad\qquad \land (\mathrm{mult}(r, \mathit{sx}, \mathit{sy}, \mathit{ms}, s_1, \{\}))
\end{array}
$$