Modular Design by Contract Visually and Formally using VCL

Nuno Amálio and Pierre Kelsen
University of Luxembourg, 6, rue Richard Coudenhove-Kalergi, L-1359 Luxembourg
{*nuno.amalio,pierre.kelsen*}*@uni.lu*

Abstract

Visual representations are widely used to describe modern-day software systems, but, in most cases, they lack rigour. This paper addresses the problems of formality, rigour and complexity in visual descriptions of software systems. It proposes a new language, VCL, that is designed to be visual, formal and modular, and that targets abstract specification at the level of requirements. VCL aims at expressing visually structural and behavioural properties of software systems. This paper presents design of VCL, outlining syntax and semantics of VCL notations of structural, behavioural, constraint, and contract diagrams, together with VCL's approach to behavioural modelling based on design by contract*. VCL's novelty lies in the fact that contracts are modular units.*

Keywords: formal modelling, visual languages, design by contract, Z

1. Introduction

Thinking, designing and communicating with pictures are recognised essential activities in traditional branches of engineering [1]. Modern day software engineering practice reflects this prominence: informal and ephemeral diagrams are used as discussion sketches; visual languages like UML are widely used to document specification and designs at different levels of abstraction [2].

Although widely used, mainstream visual languages such as the UML have several shortcomings:

- They were at large designed to be *semi-formal* (with a formal syntax, but no formal semantics). Although there have been successful formalisations of semantics for such languages (e.g subsets of UML, see [3]), they are mostly used without a formal semantics. This brings several problems: it is difficult to be precise, unambiguous and consistent, and resulting models are not mechanically analysable.
- They are not able to express diagrammatically all possible properties of software systems. This is why UML is accompanied by the textual Object Constraint Language (OCL).

To address these problems, this paper proposes the Visual Contract Language (VCL) [4], [5], [6], a visual language for abstract specification of software systems at level of requirements (or high-level designs). It presents the design of VCL, outlining syntax and semantics of VCL notations of structural, behavioural, constraint, and contract diagrams, together with VCL's approach to behavioural modelling based on *design by contract* [7].

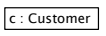
2. Syntax of VCL

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.

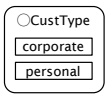
Syntax and semantics of VCL are illustrated using fragments of VCL model of secure simple Bank from [4].

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`).

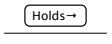


Objects are represented as rectangles; they denote an element of some set. They have a label that includes their name and may include the set to which they belong (e.g. `c` to the left).

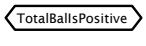


Blobs may also enclose objects, and they may be defined in terms of the things they enclose by preceding the blob's label with the symbol ◯. To the left, `CustType` is defined in this way by enumerating its elements.

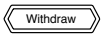
Edges connect both blobs and objects. There are two kinds: property and relational. *Property edges*, represented as labelled arrows, denote some property possessed by all elements of the set, like *attributes* in the object-oriented (OO) paradigm (e.g. `balance` to the left).



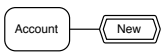
Relational edges are labelled directed lines where direction is indicated by arrow symbol above the line. Their label is within a blob because they define a set of tuples and may be inside blobs. They define or refer to some conceptual relation between blobs (*associations* in OO) – e.g. `Holds` to the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



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

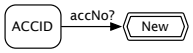


Contracts are represented as labelled double-lined hexagons. They identify operations that change state; hence, they are double-lined hexagons as opposed to single-lined constraints.



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. 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 [8] 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.

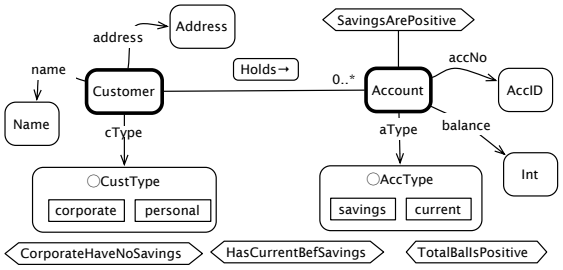


Figure 1. Structural diagram of package *Bank* in [4]

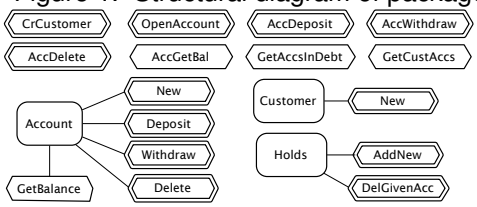


Figure 2. Behavioural diagram of package *Bank* in [4]

2.2.1. Illustration. Fig. 1 presents a well-formed SD (taken from [4], [6]). 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.

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.

Syntax of BDs used in this paper, is a subset of overall notation (package compositions constructions are not included, see [4] for details). 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*.

2.3.1. Illustration. A well-formed BD is given in Fig. 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`.

2.4. Constraint Diagrams

A VCL constraint describes a particular condition of some state of the system. They can be used to describe invariants (see [8]), and, as this paper illustrates, observe or (query) operations (operations that do not change state).

Abstract syntax of constraint diagrams (CntDs) is defined in Alloy in [8]. Syntax presented here is a subset of overall syntax (constraint expressions involving logical operators and quantifiers are not included; see [8], [4] for further details on this feature). A CntD has a *name*, a *declarations* compartment and a *predicate* compartment. Constraints 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 CntDs 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.

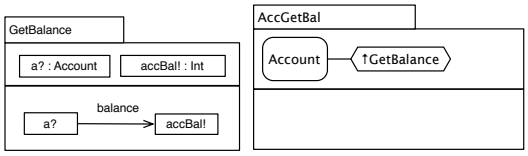


Figure 3. Constraint diagrams of operations `Account.GetBalance` and `AccGetBal` (global)

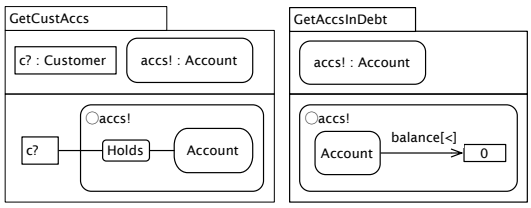


Figure 4. Constraint diagrams of global operations `GetCustAccs` and `GetAccsInDebt`

2.4.1. Illustration. Figs. 3 and 4 give examples of well-formed CntDs (from [4]). These are as follows:

Fig. 3 presents CntDs 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 `accBal!` to say that `accBal!` is to hold value of property. Global operation `AccGetBalance` does a total import (symbol ↑) of the local operation.

Fig. 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).

2.5. 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 (CctDs) 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 CntDs) for pre- and post- conditions. VCL CctDs comprise a name, a declarations compartment and a predicate compartment sub-divided into pre- (left) and post-condition (right) compartments. Figs. 5, 6 and 7 present well-formed CctDs.

Certain CctDs 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:

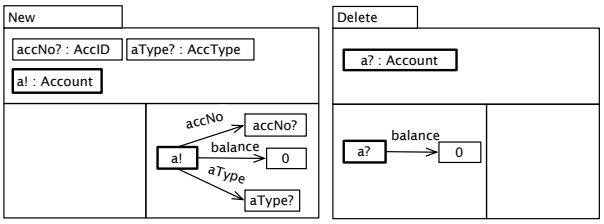


Figure 5. Contract diagrams of local operations `New` and `Delete` of blob `Account`

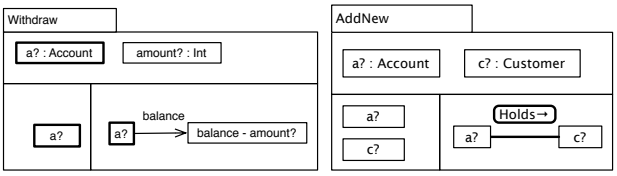


Figure 6. Contract diagrams of local operations `Account.Withdraw` and `Holds.AddNew`

- 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.
- 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 CntDs, 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 CntDs, 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 CctDs do not admit quantified expressions (whose syntax is not explained here, see [4]), user may draw separate CntDs for more complicated expressions.

2.5.1. Illustration. CctDs of Figs. 5, 6 and 7 are well-formed (from [4]). They are follows:

- Operation `New` (Fig. 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. 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. 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. 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. 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. 7, right) does a total import (symbol ↑) of local contract `Account.Withdraw`.

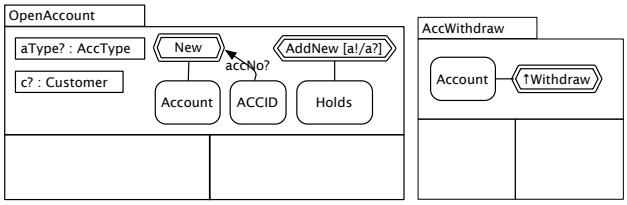


Figure 7. Contract diagrams of global operations `OpenAccount` and `AccWithdraw`

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 [9], [3], which is a semantic domain of object orientation for the language Z [10]. 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.

3.1. Structural Diagrams

SDs are mapped into ZOO following approach for construction of state spaces outlined in [9], [3]. 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 [10] (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).

3.1.1. 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. 1.

VCL blob `Int` of Fig. 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 [3] 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:

Customer

$$name : Name$$
$$address : Address$$
$$cType : CustType$$

SCustomer

$$sCustomer : \mathbb{O}\ Customer$$
$$stCustomer : (\mathbb{O}\ Customer) \pfun Customer$$
$$\dom stCustomer = sCustomer$$

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

AHolds

$$Holds : \mathbb{O}\ Customer \rel \mathbb{O}\ Account$$

Overall ensemble of structures that SD of Fig. 1 defines is defined by conjoining the definitions of blobs and relational-edges:

SystemSt

$$SCustomer;\ SAccount;\ AHolds$$

Overall system state is constrained by the system's global invariants (see Fig. 1); schema representing these are placed in predicate of overall system Z schema:

System

$$SystemSt$$

$$CorporateHaveNoSavings \land HasCurrentBefSavings$$
$$TotalBalIsPositive$$

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[1]. 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.

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 2.4, this is selected through symbol ↑. Partial importing

1. BD's package-based introduced in [4] have other meanings.

(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 2.4, 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.

3.4. Constraint Diagrams

CntDs 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 3.3.

3.4.1. Illustration. Z representation of operation `GetBalance` of Fig. 3 (left) is:

AccountGetBalance

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

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

This uses an observe promotion schema (see [9], [3]).

Z definition of `AccGetBalance` of Fig. 3 (right) is:

$$AccGetBalance == System \land SAccountGetBalance$$

Z definitions of operations of Fig. 4 are as follows:

GetCustAccounts

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

GetAccsInDebt

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

3.5. Contract Diagrams

CctDs are represented as Z schemas. They define a relation between pairs of states. They are interpreted similarly to CntDs, 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.

3.5.1. Illustration. Local operations of Figs. 5 and 6 are represented in Z as follows:

AccountNew

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

AccountDelete

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

AccountWithdraw

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

$$SAccountNew == \exists Account\ ' \bullet \Phi SAccountN \land AccountNew$$

$$SAccountDelete == \exists Account \bullet \Phi SAccountD \land AccountDelete$$

$$SAccountWithdraw == \exists \Delta Account \bullet \Phi SAccountU \land AccountWithdraw$$

HoldsAddNew

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

Global operations `OpenAccount` and `AccWithdraw` follow ZOO specification of system operations (see [9], [3]). Operation `OpenAccount` is defined in Z as:

$$\Psi OpenAccount == \Delta System \land \Xi SCustomer$$
$$OpenAccount0 == [\ c? : \mathbb{O}\ CustomerCl;\ \Delta System \mid c? \in sCustomer\ ]$$
$$ConnAccountNew == [\ accNo? : ACCID \mid accNo? \in ACCID\ ]$$
$$OpenAccount == (\Psi OpenAccount \land SAccountNew \land OpenAccount0 \land ConnAccountNew \land AHoldsAdd[a!/a?]) \setminus (accNo?, a!)$$

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

Z definition of operation `AccWithdraw` is:

$$\Psi AccWithdraw == \Delta System \land \Xi SCustomer \land \Xi AHolds$$
$$AccWithdraw == \Psi AccWithdraw \land SAccountWithdraw$$

4. Discussion

VCL. This paper outlines the syntax and semantics of VCL and introduces VCL's approach to behavioural modelling. It introduces the notations of behavioural and contract diagrams. Work presented here, together with VCL's approach to structural modelling presented in [6], and VCL's coarse-grained modularity approach based on packages presented in [4] makes design of overall VCL, a language designed for modular abstract specification of software systems at level of requirements.

Modularity. 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.

Design of VCL. 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 [4].

Verification and validation. Formal Z semantics of VCL outlined here enables verification and validation of VCL models using Z theorem provers. [11], [3] presents a visual approach to formally validate ZOO models using UML object diagrams; we intend to incorporate this approach in VCL in the future.

Usability. As discussed in [6], VCL has been designed to be well matched to meaning and to enable users to infer meaning from patterns, following usability guidelines. This can be observed in declarations, pre-conditions and post-condition compartments of contract diagrams, which closely mimic underlying structure of operations.

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).

	Total Lines of Z	From visual	Percentage of visually
VCL	490	484	98.8%
UML of [3], [9]	439	195	44.4%

Table 1. Visual expressiveness in relation to generated Z: VCL vs UML-based model of [3], [9].

VCL described all eight system operations of package *Bank* (see [4]). UML-based specification of [9], [3] needed to resort to Z to totally describe them. Table 1 compares VCL model of *Bank* package presented here with UML-based model of [9], [3] 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. 1) that could not be expressed visually and was expressed directly in Z by embedding the Z text into a CntD (see [6], [4]).

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 [12], we did not require quantification to express contracts.

Practical Value. VCL is visual and modular for practical reasons: visual representations have proved valuable in engineering [1], and modularity helps tackling complexity. VCL was applied successfully to a large case study [12]; we found that it was more productive to specify in VCL than in Z directly[2], and that the visual nature of VCL enhanced usability, readability and communication. These claims are to be subject to empirical rigorous validation (future work). Currently, we are developing tool support for VCL[3] [6] to further enhance VCL's practical value.

5. Related Work

Use of left and right compartments to mean pre- and post-conditions is inspired by Catalysis' snapshot-pairs [13], which represent specific system states. VCL contracts denote a relation between before and after states.

Several approaches represent contracts as pairs of UML object-diagrams. Lohmann et al. [14], [15], [16] translates, using graph transformation rules, UML class diagrams and contracts to Java skeletons and JML assertions. These rules are akin to the differential meaning of VCL contract diagrams. Visual OCL [17], [18] also uses graph transformations to go from contracts to OCL. Like VCL, these approaches follow a translational approach to semantics. Hausmann [19] provides a modelling technique based on graph transformation rules, which supports some degree of modularity because rules can be invoked. VCL however

2. See [12] for details; the Z expert developer found that it was more productive to specify in VCL than in Z directly; the non Z experts could learn VCL more easily than Z.

3. http://vcl.gforge.uni.lu

is at higher-level of abstraction; unlike [19], no order is prescribed on the execution of operations; VCL enables combination of rules using not only conjunction, but also disjunction and negation, closely mimicking the rules of the Z schema calculus.

Constraint diagrams [20], [21] notation has many similarities with VCL; it describes 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.

6. Conclusions and Future Work

This paper outlines the syntax and semantics of VCL notations of structural, behavioural, constraint and contract diagrams, and introduces VCL's approach to behavioural modelling. This is part of our ongoing work on VCL, a visual and formal language for modular abstract specification of software systems. Work presented here complements [6], which presents VCL's approach to structural modelling and studies expressiveness of constraint diagrams. This paper outlined formal semantics of VCL described in formal language Z; full Z specification of case study's VCL model used here to illustrate VCL is given in [4], together with complete VCL model. We are currently formalising VCL's syntax and semantic mapping, and developing VCL's tool [6].

Most relevant contribution of VCL's design presented here is its modular approach to modelling, together with modular properties of VCL's contracts and constraints. This integrates well with VCL's coarse-grained modularity mechanism based on packages presented in [4] and used heavily in [12]. To our knowledge, no other visual-contract language achieves such a level of modularity.

References

[1] E. S. Ferguson, "The mind's eye: nonverbal thought in technology," *Science*, vol. 197, no. 4306, 1977.

[2] B. Anda, K. Hansen, I. Gullesen, and H. K. Thorsen, "Experiences from introducing UML-based development in a large safety-critical project," *Empirical Software Engineering*, vol. 11, no. 4, pp. 555–581, 2006.

[3] N. Amálio, "Generative frameworks for rigorous model-driven development," Ph.D. dissertation, Dept. Computer Science, Univ. of York, 2007.

[4] N. Amálio and P. Kelsen, "The visual contract language: abstract modelling of software systems visually, formally and modularly," Univ. of Luxembourg, Tech. Rep. TR-LASSY-10-03, 2010, available at http://bit.ly/9c5YwQ.

[5] N. Amálio and P. Kelsen, "VCL, a visual language for abstract specification of software systems formally and modularly (short paper)," in *Diagrams 2010*, ser. LNAI, vol. 6170. Springer, 2010.

[6] N. Amálio, P. Kelsen, and Q. Ma, "Specifying structural properties and their constraints formally, visually and modularly using VCL," in *EMMSAD 2010*, ser. LNBIP, vol. 50. Springer, 2010, pp. 261–273.

[7] B. Meyer, "Applying "design by contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[8] N. Amálio and P. Kelsen, "The abstract syntax of structural VCL," Univ. of Luxembourg, Tech. Rep. TR-LASSY-09-02, 2009, available at http://bit.ly/d2jNty.

[9] N. Amálio, F. Polack, and S. Stepney, "An object-oriented structuring for Z based on views," in *ZB 2005*, ser. LNCS, vol. 3455. Springer, 2005, pp. 262–278.

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

[11] N. Amálio, S. Stepney, and F. Polack, "Formal proof from UML models," in *Proc. ICFEM 2004*, ser. LNCS, vol. 3308. Springer, 2004, pp. 418–433.

[12] N. Amálio, P. Kelsen, Q. Ma, and C. Glodt, "Using VCL as an aspect-oriented approach to requirements modelling," *Transactions on Aspect Oriented Software Development*, vol. VII, pp. 151–199, 2010.

[13] D. D'Souza and A. C. Wills, *Objects, Components and Frameworks with UML: the Catalysis approach*. Addison-Wesley, 1998.

[14] M. Lohmann, S. Sauer, and G. Engels, "Executable visual contracts," in *IEEE Symposium on Visual Languages and Human-Centric Computing*, 2005, pp. 63–70.

[15] G. Engels, M. Lohmann, S. Sauer, and R. Heckel, "Model-driven monitoring: an application of graph transformation for design by contract," in *ICGT 2006*, 2006.

[16] R. Heckel and M. Lohmann, "Model-driven development of reactive information systems: from graph transformation rules to JML contracts," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 2, 2007.

[17] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer, "A visualisation of OCL using collaborations," in *UML 2001*, vol. 2185, 2001, pp. 257–271.

[18] K. Ehrig and J. Winkelmann, "Model transformation from visual OCL to OCL using graph transformation," *ENTCS*, vol. 152, pp. 23–37, 2006.

[19] J. H. Hausmann, "Dynamic meta modeling: A semantics description technique for visual modeling languages," Ph.D. dissertation, University of Paderborn, 2005.

[20] A. Fish, J. Flowe, and J. Howse, "The semantics of augmented constraint diagrams," *Journal of Visual Languages and Computing*, vol. 16, pp. 541–573, 2005.

[21] J. Howse, S. Schuman, and G. Stapleton, "Diagrammatic formal specification of a configuration control platform," *ENTCS*, vol. 259, pp. 87–104, 2009.