Specifying Structural Properties and Their Constraints Formally, Visually and Modularly Using VCL

Nuno Amálio, Pierre Kelsen, and Qin Ma

University of Luxembourg, 6, r. Coudenhove-Kalergi, L-1359 Luxembourg
{nuno.amalio,pierre.kelsen,qin.ma}@uni.lu

Abstract. The value of visual representations in software engineering is widely recognised. This paper addresses the problem of formality and rigour in visual-based descriptions of software systems. It proposes a new language, VCL, designed to be visual, formal and modular, targeting abstract specification at level of requirements, and that aims at expressing visually what is not visually expressible using mainstream visual languages, such as UML. This paper presents and illustrates VCL's approach to structural modelling based on the VCL notations of structural and constraint diagrams with a case study. VCL's contributions lie in its modularity mechanisms, and the support for two alternative styles of visual constraint modelling (one closer to set theory expressions and based on Euler diagrams, the other closer to predicate calculus and based on object graphs).

Keywords: formal modelling, visual languages, Z.

1 Introduction

The value of visual representations for problem solving is widely recognised [1]. In software engineering, visual languages have been advocated for decades [2]; this importance is demonstrated in practice: visual formalisms, such as UML, are the choice when it comes to software systems modelling [3,4].

The visual formalisms that most software engineers use, such as UML, are known as semi-formal methods [5,6]; semi-formal because they were designed to 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 [5]), they are mostly used without a formal semantics. The lack of formal semantics brings numerous problems [7]: (a) it is difficult to be precise and have a good sense of what is being specified, (b) models are prone to ambiguities and inconsistencies and (c) it is not possible to semantically analyse models mechanically. Another problem is that they cannot express diagrammatically a large number of properties of software systems; this is why UML is accompanied by the textual Object Constraint Language (OCL).

I. Bider et al. (Eds.): BPMDS 2010 and EMMSAD 2010, LNBIP 50, pp. 261–273, 2010.

Formal methods, such as Z [8] and Alloy [9], embody semantically sound languages. They do not suffer from the semantic-related problems of their semi-formal counterparts. However, despite some success stories, formal methods have not been taken up by practitioners [10,5], being used only in the safety-critical niche [10]. Like others [2,11], we see in the visual and formal a promising combination to enhance the practicality and adoptability of formal techniques.

This paper presents the Visual Contract Language (VCL) [12,13]. VCL is designed for abstract specification of software systems visually, formally and modularly. Visually because visual representations favour human-processing. Formally because formality enhances precision and enables mechanical semantical analysis. Modularly because modularity helps tackling problem complexity by enabling problem decomposition.

This paper presents design of VCL for structural modelling with a case study. This is based on VCL notations of *structural* and *constraint* diagrams. The paper is as follows:

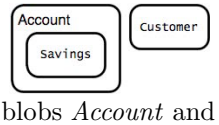
- It presents design of VCL [12,13], highlighting VCL's modularity mechanisms and its support for two alternative styles of constraint specification (one is set-theoretic and the other is akin to predicate-calculus).
- It illustrates formal semantics outline that accompanies VCL's design by giving examples of how VCL diagrams would be represented formally.
- It presents some initial results towards development of tool support for VCL[1].
- It shows how invariants, usually described textually in a formal language (such as OCL) in UML-based models, can be described visually using VCL.

2 Overview of Structural VCL

VCL has been designed to have a minimal set of visual primitives. Because these primitives are used in different types of diagrams and in different contexts, they have a core meaning that varies slightly with the context. In VCL presented here, same visual concept can be used in both structural and constraint diagrams.

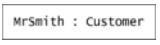
The abstract syntax of VCL notations of structural and constraint diagrams presented here is given in [14]; it is defined formally using OO *metamodels* specified in the Alloy modelling language [9].

2.1 Visual Primitives

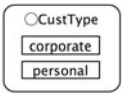


VCL's *blob* concept is like an Euler circle: a rounded contour denoting a set. Topological notions of *enclosure* and *exclusion* represent subset and disjoint relations. To the left, blobs *Account* and *Customer* represent disjoint sets of objects; *Savings* is a subset of *Account*.

[1] The *visual contract builder tool*, `http://vcl.gforge.uni.lu`



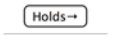
VCL's concept of an *object* or *atom* is represented as a rectangle. Objects denote an element of some set. To the left, *MrSmit* is an object of blob *Customer*.



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

Edges connect both blobs and objects to define various kinds of relations. There are two kinds of edges: property and relational.

Property edges, represented visually as directed arrows labelled with a name, denote or refer to some property possessed by all elements of the set (e.g. *balance* to the left); they are like *class attributes* in the object-oriented (OO) paradigm.



Relational edges are represented as labelled directed lines, where direction is indicated by arrow symbol above the line. Their label is within a blob because they denote a set of tuples and may be placed inside blobs. Relational edges define or refer to some conceptual relation between blobs (*associations* in OO)[2] (e.g. *Holds* to the left).

To indicate that some model structure(s) are subject to constraints, VCL uses *constraints* (e.g. *TotalBalIsPositive* to the left), which are labelled with the constraint name they refer to.

2.2 Structural Diagrams

Structural diagrams (SDs) define the structures that make the system's state space. They describe main problem domain concepts as blobs, their internal state as property edges, their conceptual relations as relational edges, and invariants as constraint references (see Fig. 1 for an example).

In SDs, there are two types of blobs: *domain* and *value*. Domain blobs, represented using a bold line, are part of the state of overall system; they are dynamic and need to be maintained. Value blobs define an immutable set of values that do not need to be maintained. In Fig. 1, *Account* and *Customer* are domain blobs; *Name* is a value blob. In SDs, blobs may be defined by enumerating its constituent objects; blobs *CustType* and *AccType* are defined in this way.

2.3 Constraint Diagrams

Constraint diagrams (CntDs) are made of three compartments: *name*, *declarations* and *predicate* (see Fig. 2 for an example). The declarations compartment introduces variable names together with other constraints being imported. The predicate compartment actually defines the constraint. A predicate can be formed of objects, blobs, relational and property edges as in CntD *AccSavings*

[2] Relational edges denote a relation between sets or a tuple depending on whether they connect blobs or objects.

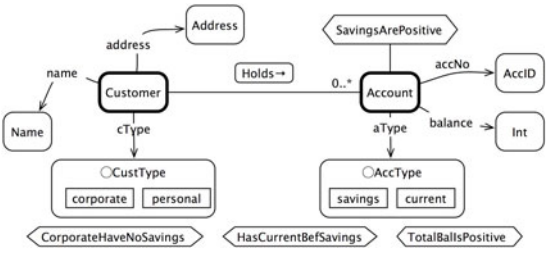


Fig. 1. Structural diagram of simple Bank

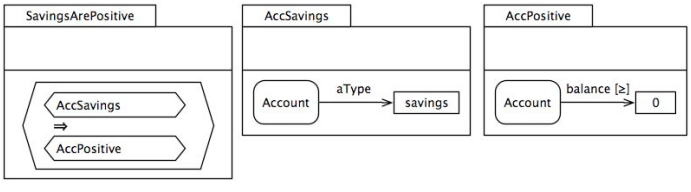


Fig. 2. Constraint diagrams for *Account* invariant *SavingsArePositive*

of Fig. 2 (p. 264), or made up of a constraint reference expression as in CntD *SavingsArePositive* of Fig. 2. VCL enables combination of constraint diagrams using logical operators, namely: negation, conjunction, disjunction, implication, and universal and existential quantification.

2.4 Semantics

VCL's design presented here is accompanied by an outline of its formal semantics. VCL embodies a *generative* (or translational) approach to semantics. It is to be used together with a textual formal specification language, *target language*, that sits in the background and a target language semantic model. The semantics of a VCL specification is the generated target language specification. This paper uses Z as target language and ZOO [15,5] as semantic domain. ZOO is an abstract OO semantic domain for language Z. We use this way our previous result, enabling us to focus on the visual aspects of VCL.

The VCL diagrams are mapped into a ZOO model, which comprises Z structures representing the various elements of a VCL model. Semantically, a blob is a set, property edges are properties shared by all objects of the set, relational edges are relations between sets, ensembles are collections of sets and relations, constraints are predicates that restrict some state structure or ensemble. Various VCL model elements are represented as Z schemas that can be combined in various ways.

3 Running Example

VCL is illustrated here with simple Bank case study, which is also used to illustrate the ZOO semantic domain in [15,5]. The case study's requirements are given in table 1.

The following presents VCL's structural and constraint diagrams describing *simple Bank*. The outline of VCL's Z semantics is illustrated here by presenting how VCL diagrams would be represented in ZOO. Full Z specification resulting from VCL semantics outlined here is available online [16].

Table 1. Requirements of the simple Bank system

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	To open a savings account, customer must already hold a current account with the Bank.

4 Defining the Structures That Make the State Space

VCL SDs define state structures and identify the invariants that constrain them. There are two types of invariants. *Local invariants* are attached to some blob and they affect and are described in the scope of associated blob; they are known as class invariants in OO paradigm. *Global invariants* affect and are described in the scope of an ensemble of state structures as defined by some SD.

4.1 Simple Bank System

Figure 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 (symbol ◯). *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 invariants are global: *CorporateHaveNoSavings* (Requirement $R6$), *HasCurrentBefSavings* (Requirement $R7$) and *TotalBalIsPositive* (Requirement $R5$).

4.2 Z Representation

Z representation of SDs follows ZOO approach for construction of state spaces outlined in [15,5]. Value blobs that are not enumerations are defined as given sets, enumerations as free types, domain blobs as promoted abstract data types (ADTs), and relational edges as Z relations. Finally, the ensemble of state structures defined by overall SD is built as a conjunction of those Z schemas representing domain blobs and relational edges. The following gives Z definitions of blobs *Name*, *Address*, *CustType* and *Customer*, relational edge *Holds* and state of ensemble for SD of Fig. 1.

$$[Name, Address]$$

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

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

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

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

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

5 Constraining the State Space

VCL's constraint diagrams enable specification of constraints in two styles. One is close to set theory and is based on blob constructions such as insideness and shading. The other is closer to predicate calculus and is based on object graphs.

As the examples given below show, CntDs are modules that can be composed in various ways.

5.1 Defining Constraints with Blobs

Blobs introduced in a SD are the building blocks of a VCL model. From them, derived blobs are defined for the purpose of constraining the state space. In addition to the blob relations of inclusion and exclusion, in CntDs blobs can be shaded to say that the denoted set must be empty.

Invariant *SavingsArePositive***.** This local invariant is described in Fig. 2 using three CntDs. All declarations compartments are empty because no extra declarations of names are required to describe the constraint. Invariant is described as follows:

- CntD *AccSavings* defines a predicate describing all those objects of *Account* whose property *aType* is equal to value *savings*.
- CntD *AccPositive* defines a predicate describing all those objects of *Account* whose *balance* must be greater or equal to 0 [3].

[3] 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 $\geq$.

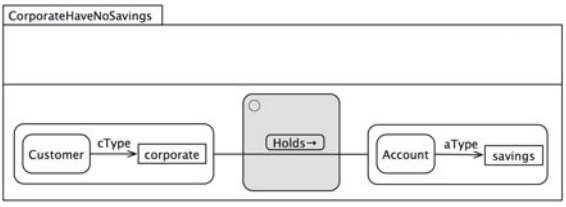


Fig. 3. Constraint diagram for invariant *CorporateHaveNoSavings*

– Finally, CntD *SavingsArePositive* defines actual constraint by saying CntD *AccSavings* implies *AccPositive*, which means that predicates encapsulated by these CntDs are being related using logical implication.

Z representation of this invariant comprises a Z schema for each CntD:

$$AccSavings \;==\; [\; Account \mid aType = savings \;]$$
$$AccPositive \;==\; [\; Account \mid balance \geq 0 \;]$$
$$SavingsArePositive \;==\; [\; Account \;\mid\; AccSavings \;\Rightarrow\; AccPositive \;]$$

Invariant *CorporateHaveNoSavings***.** This invariant is described in Fig. 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*. Shadded blob in the middle captures relation *Holds* restricted to those tuples with corporate customers and savings accounts; shading says that set must be empty, giving required meaning.

Z representation of this invariant is as follows:

CorporateHaveNoSavings

$$BankSt$$

$$\{oC : sCustomer \mid (stCustomer\ oC).cType = corporate\} \lhd Holds \rhd \{oA : sAccount \mid (stAccount\ oA).aType = savings\} = \varnothing$$

Invariant *HasCurrentBefSavings***.** This invariant is described in Fig. 4 using three CntDs. CntD *CustsWithCurrentDef* defines set of customers with current accounts (*CustCurr*). CntD *CustsWithSavingsDef* defines set of customers with savings accounts (*CustSav*). Finally, CntD *HasCurrentBefSavings* says *CustSav* is subset of *CustCurr*; these names refer to same object in the different diagrams.

Constraint importing results in importing of names. When an imported name is not explicitly declared, then it is hidden. In *HasCurrentBefSavings* *CustSav* and *CustCurr* are not declared, so they are hidden. Note the use of *insideness* property of blobs to capture domain of relation *Holds*. Blobs *CustsSav* and *CustsCurr* are defined (symbol ◯) by having inside the blobs representing the relation and set that is in domain of relation; this means that we are capturing the domain of relation *Holds* subject to restrictions as defined in constraint. See [16] for Z definition of this invariant.

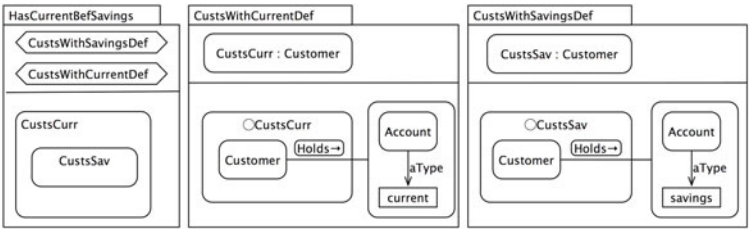


Fig. 4. Constraint diagram for invariant *HasCurrentBefSavings*

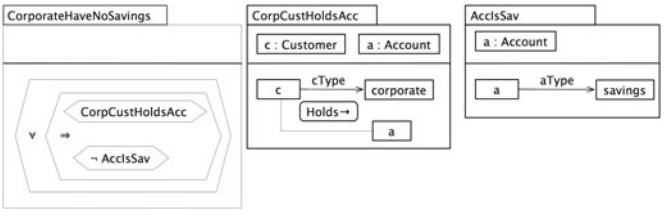


Fig. 5. Constraint diagram of *CorporateHaveNoSavings* using objects

5.2 Expressing Constraints with Objects and Quantifiers

Following CntDs illustrate use of objects and quantifiers to express constraints in a predicate-calculus style.

Invariant *CorporateHaveNoSavings*. Figure 5 gives an alternative formulation of this invariant to that given in Fig. 3, which is formulated using blobs. This defines CntDs *CorpCustHoldsAcc* (customer c is of type *corporate* and holds account a) and *AccIsSav* (account a is of type *savings*). CntD *CorporateHaveNoSavings* then says that the former implies the negation of the latter to say that a customer of type corporate must not have a savings account. Universal quantifier asserts that implication must hold for all customers c and accounts a. All variables are bound by the quantifier in both diagrams; name a in two different diagrams refers to same object. See [16] for Z definition of this invariant.

Invariant *HasCurrentBefSavings*. Figure 6 expresses this constraint by saying that all customers having a savings must also have a current account. A quantifier applied to a constraint binds all its variables, except when a variable has its scope extended by a *communication edge*. In Fig. 6, c's scope is extended in this way; hence, it is not bound by the two existential quantifiers. It is, however, bound by universal quantifier in *HasCurrentBefSavings*. See [16] for Z definition of this invariant.

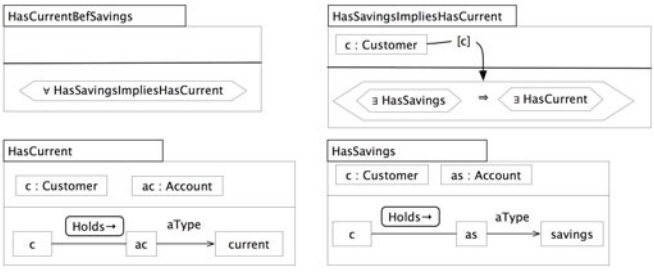


Fig. 6. Constraint diagram of *HasCurrentBefSavings* expressed using objects

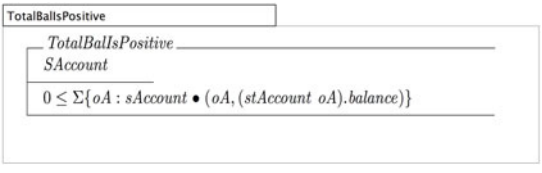


Fig. 7. Z constraint *TotalBalanceIsPositive* embedded in VCL constraint diagram

5.3 Constraints That Cannot Be Expressed Visually

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. Assuming a sum operator defined in the target language toolkit (see [15] for details), constraint *TotalBalanceIsPositive* (Fig. 7) is expressed in Z; its text is embedded in VCL CntD.

6 Discussion

VCL and our previous work. This paper presents part of our ongoing work on VCL, a visual language for abstract specification of software systems. VCL uses our previous result, ZOO [15,5], a semantic domain of object-orientation expressed in language Z, which is well studied; it has been applied to several case studies published in the literature. This enables us to focus on the visual aspects of VCL; a result of work presented here is that we can describe visually structures that previously could only be described textually in Z.

Use of Alloy. Metamodels of VCL notations presented here were formally defined in Alloy (see [14]), and refined into concrete syntax metamodels implemented in VCL's *visual contract builder* tool[4] (an Eclipse plug-in based on GMF framework[5]). Alloy was of great help in defining VCL's syntax: (a) it enabled

[4] `http://vcl.gforge.uni.lu`
[5] `http://www.eclipse.org/modeling/gmf/`

Table 2. Comparison of visual expressiveness in relation to generated Z between VCL model of case study's *Bank* package and UML-based model of [15,5]

	Total Lines of Z	**From visual**	**Percentage of visually**
With VCL	263	257	97.7%
With UML as in [15]	215	195	91%

precise expression of well-formedness constraints; (b) its model-finder and visualisation features helped in understanding VCL's syntax; (c) its model-checking feature verified satisfaction of certain desired properties; and (d) OO structure of its models meant a smooth transition from abstract to concrete syntax (VCL's tool uses OO metamodel-based technologies to construct graphical parsers).

Usability. This has been a concern guiding VCL's design:

- VCL's visual concepts are designed to be well-matched to meaning and give good sense of their mathematical underpinnings (*closeness of mapping* guideline of [17]). VCL's blob symbol, for instance, a circular contour denoting a set, is a well-known mathematical visual concept (as Venn or Euler circles).
- 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 [17]).

Expressiveness. VCL is designed to enable precise and rigorous abstract specification and to express visually constraints not visually expressible in UML. VCL's design is accompanied by an outline of a formal Z semantics, which has been illustrated here with examples; Z model representing semantics of case study's VCL model presented here is given in [16].

VCL was able to express visually 3, out of a total of 4, system invariants of case study; UML-based description of [15,5] describes none of them. Table 2 compares number of lines of generated Z for VCL specification presented here, and UML-based description of [15,5]. 97.7% of case study could be expressed visually using VCL; remaining 1.5% (constraint *TotalBalanceIsPositive*, above) must be expressed textually. This gives a 6.7% increase from [15,5][6].

Modularity. VCL examples given in this paper highlight VCL's modularity and abstraction mechanisms. The constraint visual primitive abstracts away from the details of constraint definitions in CntDs. CntDs are modules that can be composed in various ways. For example, invariant *HasCurrentBefSavings* described in Fig. 4 is defined using two auxiliary CntDs, *CustsWithCurrentDef* and *CustsWithSavingsDef*, which are combined in CntD *HasCurrentBefSavings* through importing. The same auxiliary CntDs could be used to state other constraints which require the set of customers with a current account account, and the set of customers with a savings account. Invariant *CorporateHaveNoSavings*

[6] This increases to 54.4% with VCL's language of contracts; [15,5] describes very few behaviour visually.

of Fig. 5 illustrates how CntDs can be combined using logical operators, such as universal quantification and implication.

Two Modelling Styles. VCL enables two styles of constraint specification. Blob constructions enable specification based on sets and relations. Object-based constructions closely mimic predicate calculus formulas. Using either style is often a personal choice, but some solutions call for blobs, others for objects. In our experience, blob constraints tend to be more compact, but users familiar with predicate calculus find object expressions easier to follow. For instance, blob constraint of Fig. 3 (p. 267), is more compact than semantically equivalent object constraint of Fig. 5 (p. 268) because it uses fewer modelling elements. On the other hand, object constraint of Fig. 6 (p. 269) is more readable than that of Fig. 4 (p. 268) by those more familiar with predicate-calculus. Semantically, blob constraints tend to result in more concise and compact Z expressions.

Practical Value. We design a language for visual expression because, as argued in [1,2], there is value in them. VCL has been applied to a case study of a large-scale system [18]. We found that it was more productive to specify in VCL than in Z directly, and that VCL enhanced usability and readability of resulting specification.

7 Related Work

Evans et al [7] propose to define formally UML's semantics in Z; intent is to work on UML realm only. VCL's semantic approach generates a working Z model, which can be used for proof and animation by Z experts, and to support model analysis assisted by diagrams as proposed in [19,5]. As shown with case study, Z is also used to *augment* visual description and express what can not be expressed visually (constraint *TotalBalIsPositive*, Fig. 7, p.269), and VCL is able to express visually what was not visually expressible with UML.

Several approaches propose visual constraint notations to eliminate or minimise need for textual languages like OCL. These fall into two groups depending on supported style of constraint specification: *sets* and *predicate-calculus*. Constraint or spider diagrams [20,21], like VCL's blob constructions, are akin to Euler diagrams in that they express set-based constraints (inclusion, intersection, etc). Visual OCL [22] and Story Decision Patterns [23] have a semantics based on graph-transformation; they result in constraints akin to predicate calculus like VCL's object constraints. To our knowledge, VCL is the only visual language that integrates both styles of constraint specification. However, these languages are more mature than VCL, which still lacks a complete formal definition.

Another prominent feature of VCL is its support for modularity. Constraint diagrams [20,21] lack mechanisms to compose constraints modularly similarly to the VCL constraint composition mechanisms illustrated here. Visual OCL [24,22], like VCL, also provide logical operators and quantifiers, and a way of composing constraints, but does does not support set-based constraints; also

Visual OCL has more visual concepts than VCL, which does not favour usability. Story Decision Diagrams [23] notation has modular operators and quantifiers. In terms of expressability, [23] is close to our object language; when trying to express the constraints of [23] in VCL, structures used in both solutions were close to each other. However, VCL's syntax is closer to predicate calculus than [23] — VCL's logical operators are standard implication, conjunction, negation and disjunction (it isn't so in [23]) —, and so it benefits from engineers' familiarity with predicate calculus, and VCL enables specification of set-based constraints.

Our work is influenced by Harel's *Higraphs* [2], which are based on Euler diagrams and are basis of *statecharts*. From [2], we borrowed the *blob* and took inspiration for both language of VCL SDs and blob-based constructions of CntDs.

8 Conclusions

This paper presents some results regarding our ongoing work on VCL, a visual and formal language for abstract specification of software systems. It presents design of VCL's structural and constraint diagrams with a case study, and illustrates outline of VCL's formal Z semantics that accompanies VCL's design presented here by showing how VCL diagrams would be represented in Z.

VCL presented here is just a design of a language. This design includes an outline of the language's formal Z semantics. We intend to have a complete formal definition of VCL. Currently, we are working on defining formally the *semantic mapping*, from syntax to Z semantics illustrated here, which will be the basis of automatic generation of Z models in VCL's tool.

This paper demonstrates VCL's modularity at the level of constraints and its capability at expressing visually constraints not expressible visually in UML. It shows that VCL was able to express more visually than a UML-based approach for the case study used here. VCL is able to describe 3 out of 4 system invariants; UML describes none of them. The paper illustrates two styles of visual constraint specification: one is set-theoretic, the other is akin to predicate calculus.

We are working on a coarse-grained modularity mechanism of packages to enable separation of concerns at the requirements level [12]. We have successfully applied this mechanism to tackle complexity of a large-scale case study in [18].

There are several aspects in the work presented here that are, to our knowledge, novel. The modularity of VCL's approach to constraint specification is something not much explored in this area. Perhaps, the most relevant novelty is that VCL enables the specification of constraints visually in both set-theoretic and predicate-calculus styles. To our knowledge, no one integrated in a single constraint language Euler-like diagrams with object graphs used in graph transformation approaches.

References

1. Larkin, J.H., Simon, H.A.: Why a diagram is (sometimes) worth ten thousand words. Cognitive Science 11, 65–99 (1987)
2. Harel, D.: On visual formalisms. Commun. of the ACM 31(5), 514–530 (1988)

3. Sumner, M., Sitek, J.: Are structured methods for system analysis and design being used. J. of Systems Management 37(6), 18–23 (1986)
4. Anda, B., Hansen, K., Gullesen, I., Thorsen, H.K.: Experiences from introducing UML-based development in a large safety-critical project. Empirical Software Engineering 11(4), 555–581 (2006)
5. Amalio, N.: Generative frameworks for rigorous model-driven development. Ph.D. thesis, Dept. Computer Science, Univ. of York (2007)
6. Amálio, N., Polack, F., Stepney, S.: Frameworks based on templates for rigorous model-driven development. ENTCS 191, 3–23 (2007)
7. Evans, A., France, R.B., Lano, K., Rumpe, B.: The UML as a formal modelling notation. In: Bézivin, J., Muller, P.-A. (eds.) UML 1998. LNCS, vol. 1618, pp. 336–348. Springer, Heidelberg (1999)
8. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. PH (1996)
9. Jackson, D.: Software Abstractions: logic, language, and analysis. MIT Press, Cambridge (2006)
10. Cleland, G., MacKenzie, D.: Inhibiting factors, market structure and industrial uptake of formal methods. In: WIFT 1995, pp. 46–60. IEEE, Los Alamitos (1995)
11. Harel, D.: Biting the silver bullet: Toward a brighter future for system development. Computer 25(1), 8–20 (1992)
12. Amálio, N., Kelsen, P., Ma, Q.: The visual contract language: abstract modelling of software systems visually, formally and modularly. Tech. Report TR-LASSY-10-03, Univ. of Luxembourg (2010), `http://bit.ly/9c5YwQ`
13. Amálio, N., Kelsen, P.: VCL, a visual language for modelling software systems formally. In: Diagrams 2010. LNCS. Springer, Heidelberg (2010)
14. Amálio, N., Kelsen, P.: The abstract syntax of structural VCL. Tech. Report TR-LASSY-09-02, Univ. of Luxembourg (2009), `http://bit.ly/4DHWky`
15. Amálio, N., Polack, F., Stepney, S.: An object-oriented structuring for Z based on views. In: Treharne, H., King, S., C. Henson, M., Schneider, S. (eds.) ZB 2005. LNCS, vol. 3455, pp. 262–278. Springer, Heidelberg (2005)
16. Amalio, N.: ZOO specification of VCL model describing structural aspects of simple bank case study (2010), `http://bit.ly/4yBrsW`
17. Green, T.R.G., Petre, M.: Usability analysis of visual programming environments: a 'cognitive dimensions' framework. J. of Visual Languages and Computing 7(2), 131–174 (1996)
18. Amálio, N., Ma, Q., Glodt, C., Kelsen, P.V.: specification of the car-crash crisis management system. Tech. Report TR-LASSY-09-03, University of Luxembourg (2009), `http://vcl.gforge.uni.lu/doc/vcl-cccms.pdf`
19. Amálio, N., Stepney, S., Polack, F.: Formal proof from UML models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 418–433. Springer, Heidelberg (2004)
20. Kent, S.: Constraint diagrams: Visualizing invariants in object-oriented methods. In: Proc. of OOPSLA 1997, pp. 327–341. ACM Press, New York (1997)
21. Fish, A., Flower, J., House, J.: The semantics of augmented constraint diagrams. J. of Visual Languages and Computing 16, 541–573 (2000)
22. Ehrig, K., Winkelmann, J.: Model transformation from visual OCL to OCL using graph transformation. ENTCS 152, 23–37 (2006)
23. Giese, H., Klein, F.: Beyond story patterns: Story decision diagrams. In: Proc. of Fujaba Days 2006, pp. 2–9 (2006)
24. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: A visualisation of OCL using collaborations. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 257–271. Springer, Heidelberg (2001)