Electronic Communications of the EASST
Volume 31 (2010)

Proceedings of the Second International Workshop on Visual Formalisms for Patterns (VFfP 2010)

Visual Specification Patterns

Andrew Fish, Ali Hamie, John Howse

14 pages

Guest Editors: Paolo Bottoni, Esther Guerra, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Visual Specification Patterns

Andrew Fish, Ali Hamie, John Howse

University of Brighton, UK

Abstract: Visual modelling notations such as *constraint diagrams* can be used for the behavioural specifications of software components. This includes specifying invariants on classes or types and preconditions and postconditions of operations. However, one current problem in specifying components comes from the fact that editing constraints manually is time consuming and error prone and so we may adopt a pattern-based approach to alleviate this problem. One way to simplify the definition of constraints is to identify and capture those recurring constraints in the form of visual specification patterns. Such patterns would facilitate the automatic generation of diagrammatic constraints. This paper identifies some specification patterns that frequently occur when specifying software components and provides a diagrammatic representation of these patterns. This will form the basis of a library of specification patterns that could be used in the context of tools. We also show how such patterns can be combined in order to specify more complex constraints.

Keywords: Formal specification, constraint diagrams, visual formalisms

1 Introduction

Component-Based Development or CBD is a software development approach where software applications are built using components, and these components can come from a number of different sources, be written in several different programming languages, etc. By employing such an approach one can improve the efficiency and quality of software development and increase the flexibility of the resulting software systems [SGM02]. An essential prerequisite towards achieving the goal of CBD is an appropriate and standardized specification of software components. Visual modelling notations such as *constraint diagrams* [Ken97] can be used for specifying the behavioral aspects of software components and their constraints. Constraint diagrams are a formal diagrammatic language that can be used for describing invariants on classes as well as preconditions and postconditions of operations. However, developing constraint specifications for software components is time consuming and an often error prone task. This is because typical specifications may contain numerous constraints, which in addition often state complex facts about the elements of the component's model. Ackermann [Ack05a, Ack05b, AT06] proposes a solution to this problem based on the idea of specification patterns from which OCL [WK03] constraints can be automatically generated. Nine patterns that frequently occur in the behavioural specifications of software components have been identified in [Ack05b].

To simplify constraint definition we follow a similar approach by using specification patterns from which visual constraints can be generated automatically. This applies in the context of specifying components in a diagrammatic way. As a first step towards achieving this goal we identify a list of frequently occurring specification patterns and describe them in some detail. The

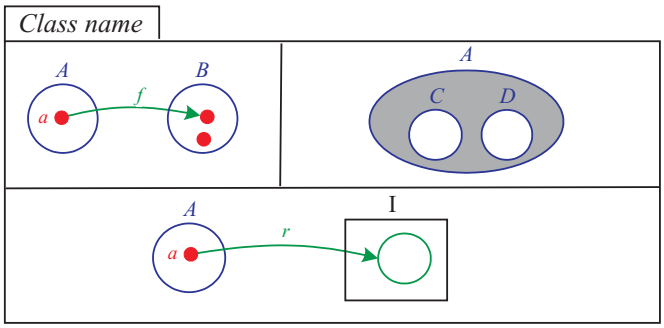

Figure 1: An invariant

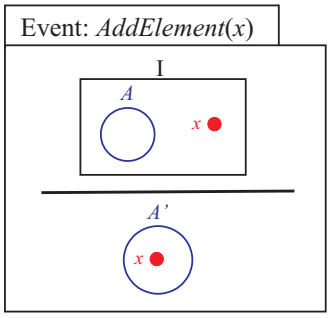

Figure 2: Event specification

main contributions of this paper are: the proposal to use visual specification patterns to simplify component specifications, the identification of frequently occurring specification patterns within the modelling framework and the development of a description scheme to characterize those patterns. The OCL specification for the identified patterns will also be included alongside the diagrammatic specification.

2 The Diagrammatic Framework

A class is modeled in terms of an invariant and its operations, specified as pre/post condition contracts. The diagram in Figure 1 is an example of an invariant. This diagram consists of three sub-diagrams. The closed curves (circles, ellipses, rectangles) represent sets. In each sub-diagram, the curves form an Euler diagram and their spatial relationships express semantic relationships between the sets: non-overlapping curves assert that the sets are disjoint; a curve placed inside another asserts a subset relationship. We use the convention that labelled rectangles represent types. The dots are called *spiders*. Unlabeled spiders assert the existence of elements in the sets represented by the regions of the diagram in which they are placed. The labeled spiders in this diagram are acting as free variables. An *arrow* represents a binary relation, where its *source* and *target* may be either a spider or a curve; its target then corresponds to the image of its source under a relation identified by the label on that arrow.

In the first sub-diagram there are two disjoint sets A and B. The spider labeled a is a free variable and is the source of the arrow f, while the target of f is a spider in B; thus f represents a function mapping each element of A to an element in B. Different spiders represent distinct elements, so there is an element in B that is not the image of a under f. In the second sub-diagram C and D are disjoint sets and each is a subset of A. As there are no spiders in it, the shaded region represents the empty set. So this sub-diagram asserts that A is partitioned into subsets C and D. In the third sub-diagram the rectangle represent a type. The arrow labeled r represents a relation as it maps each element of A to a subset of I. The semantics of the sub-diagrams are conjoined to give the semantics of the diagram.

Operations are divided into queries and events. Queries specify operations that may be ap-

plied to an object of a class leaving its state unchanged, whereas events specify their allowable changes-of-state. Either may have input- or output-arguments that are declared in its pre-condition. Each event also has a post-condition, wherein 'dashed' names denote values of the corresponding variable after any occurrence of the event, adapting the same convention from Z, and playing the role of the @pre operator in OCL.

The diagram in Figure 2 specifies an event for adding an element to a set. This event is specified in terms of a pre-condition and a post-condition. The pre-condition is specified above the line and the post-condition below the line. The diagram can be interpreted as 'if the conditions above the line hold then the conditions below the line hold after an occurrence of the event'. In the pre-condition x (which is of type I) is not in A and in the post-condition x is in A', the updated version of A. So this event adds x to A. This framework uses the convention 'the rest remains unchanged' [SPB90] to allow operation constraints to be presented in concise from. More details about the framework and the diagrammatic approach can be found in [HS05, FFH05].

Some of the benefits of diagrammatic notations are evident in the diagrams we have seen so far, where both set intersection, disjointness and containment are represented visually. These diagrams have properties that are thought to correlate with areas where diagrams are superior to symbolic notations, from a user interpretation perspective, because they are well-matched to their set-theoretic semantics [Gur01]. Extending this observation, using containment to represent set inclusion has the added benefit that the transitive property of the (semantic) subset relation is mirrored by the transitive property of (syntactic) containment.

3 Diagrammatic Specifications and Visual Specification Patterns

Most approaches to component specification recommend the use of formal mathematical notations since they enable a common understanding of specification results across different developers and companies. The use of formal methods, however, is not undisputed. Some authors argue that the required effort is too high and the intelligibility of the specification results is too low - for a discussion of advantages and liabilities of formal methods see [Hal90].

The use of visual notations for specifying components has the advantage of being more intuitive and accessible to developers than formal mathematical notations that are based on set theory and predicate logic. Despite their advantages visual modelling notations cannot solve all problems associated with the use of formal methods. Writing and editing constraint diagrams manually can sometimes be time consuming and error-prone. According to Ackermann [Ack05a], the same problem is encountered when using OCL to specify software components. Similar experiences were made by other authors that use OCL constraints in specifications (outside the component area) [LDF04, HJR02]. They conclude that it takes a considerable effort to master OCL and use it effectively.

However, behavioral aspects have great importance for component specifications. For example, the specification of components within a video rental service case study have filled many pages and required significant effort. For component specifications to be practical it is therefore indispensable to simplify the diagrammatic-based behavioral specification.

A strategy to simplify diagrammatic specifications and reduce errors include better tool support and the use of predefined visual specifications patterns. The latter approach seems to be

particularly promising since the analysis of the video rental service case study reveals that most of the constraints can be derived from a few, frequently occurring, specification patterns. An example of such a pattern is the following: An attribute of a class in the specification type diagram is unique for each instance. That is the attribute plays the semantic role of a key.

The way to use such patterns in the specification is as follows. Assume that the modeler needs to describe a certain behavioural condition in the diagrammatic notation. First he checks the library of predefined visual specification patterns which is part of his specification tool and finds a matching one. Once a suitable pattern is found the modeler does not need to write/draw it manually but instead he selects the pattern and provides the model elements for which the pattern shall be applied. The tool then checks his input for consistency and generates the constraint.

The advantage of this approach is that the specification process is simplified because the specification is generated automatically, is less error-prone and requires less expertise in the visual modelling notation. Moreover, when the patterns are well-known, it will be enough to specify a pattern (without the generated diagrams) allowing the user to recognize the constraint more easily.

In order to achieve this goal we proceed as follows. First, visual constraints that frequently occur in component specifications are identified. Then we develop a description technique that enables the description of those patterns in such a way that they can be adapted to special cases and automatically generated into diagrammatic constraints. Lastly, the description technique is applied to the identified patterns so that a generator can be implemented for each pattern. Any newly identified patterns can easily be included in the pattern library at a later time.

4 Frequently Occurring Specification Patterns

In this section we present a list of patterns that frequently occur in the behavioral specification of software components. These patterns will be presented in diagrammatic form using constraint diagrams. In order to identify such patterns several case studies were considered, including a video rental service [HS05], library systems, medical information models and a variety of collected specification examples.

4.1 Description Scheme for Specification Patterns

In this section we provide a description of some specification patterns. This includes all of the relevant details of a pattern presented in a structured and uniform way. The first characteristic is the pattern name that identifies the pattern and serves as a short semantic explanation.

All patterns have one or more parameters that allow the adaptation of the pattern to specific contexts. The pattern lists these parameters together with their types unless it is clear from the context in which case the type is omitted. Parameter names are a means of matching up diagrammatic entities. Parameters can be of elementary type (like String) or are elements from the diagrammatic notation. For example, a parameter could be a curve representing a set or an arrow representing an association or relation. Type parameters are omitted when the pattern applies for any type. The description can also document the restrictions on the application of the pattern, such as the conditions that the pattern parameters must fulfill. For example, for a pattern with

Figure 3: Binary partition

parameters *op* (of type Operation) and *par* (of type Parameter) it might be required that *par* is a parameter of *op*.

4.2 Constraint Patterns in Invariants

In this section we consider patterns that frequently occur when specifying invariants on types or components.

Pattern 1: Binary Partition

The binary partition constraint is very common (Figure 3). It partitions a given set A into two subsets B and C. The name of the pattern is *Binary Partition*, and the sets A, B, and C are the parameters of the pattern. The pattern constraint is an invariant that is valid for the input parameters. The constraint asserts that sets B and C partition set A (although the sets B and C can be empty).

Examples of this pattern can be found when specifying medical information and library systems. Applying the pattern to these systems yields the invariants shown in the two diagrams on the right hand side of Figure 3. The diagram on the left is an instantiation of $BinaryPartition(A,B,C)$ with parameter A instantiated as *Patient*, B as *Alive* and C as *Dead* and asserts that a patient is either dead or alive (but not both). The diagram on the right is an instantiation of $BinaryPartition(Title,InColl,ExColl)$ and asserts that a title is in a library's collection (*InColl*) or not in the collection (*ExColl*). The binary partition constraint generalizes to any set partition.

In OCL, this constraint (ignoring the context) can be stated as follows:

$$inv : A = B \rightarrow union(C) \; and \; B \rightarrow intersection(C) = Set\{\}$$

where *union* and *intersection* are the OCL operations for set union and set intersection respectively. The $\rightarrow$ indicates applying an operation on the whole set or collection. The keyword *inv* indicates that the constraint is an invariant.

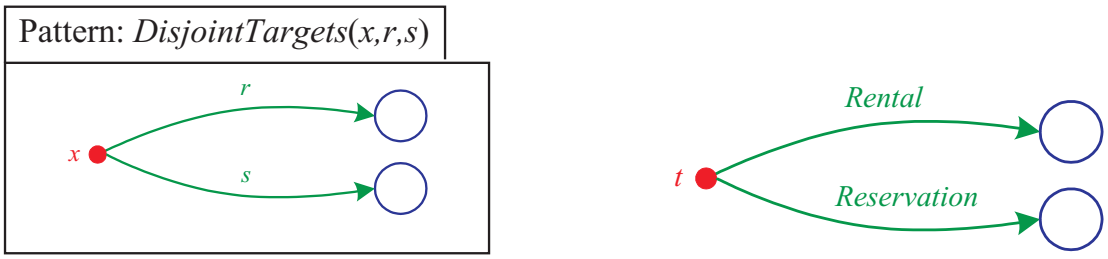

Figure 4: Disjoint targets

Pattern 2: Disjoint targets

The pattern shown in Figure 4 is also very common, and asserts that the relational images of some value under two relations are disjoint. The name of the pattern is $DisjointTargets$, and the parameters are the value x and the relations r and s. The constraint asserts that the relational images of x under r and s yields disjoint sets and that these sets do not contain x.

The diagram on the right hand side of Figure 4 shows an instantiation of the pattern which comes from a video rental store specification. It is $DisjointTargets(t, Rental, Reservation)$ and asserts that the rentals and reservations of (title) t are disjoint, indicating that no video title can be both rented and reserved (by the same member) at the same time. The set of members renting title t is given by the closed curve at the end of the arrow labeled *Rental*; this set is seen to be disjoint from the set of members reserving t.

In OCL, this pattern is common and occurs when navigating associations on a class diagram. Suppose that r and s represent role names of two associations between class A and class B with multiplicity $*$ at the B's end. Let x be an object of class A, then $x.r$ and $x.s$ are two subsets of B denoting the relational images of r and s respectively. The constraint is specified in OCL as follows:

$$inv : x.r \rightarrow intersection(x.s) = Set\{\} \; and \; not(x.r \rightarrow includes(x)) \; and \; not(x.s \rightarrow includes(x))$$

In the context of OCL, the parameters of the pattern include the classes A and B, and the association roles r and s.

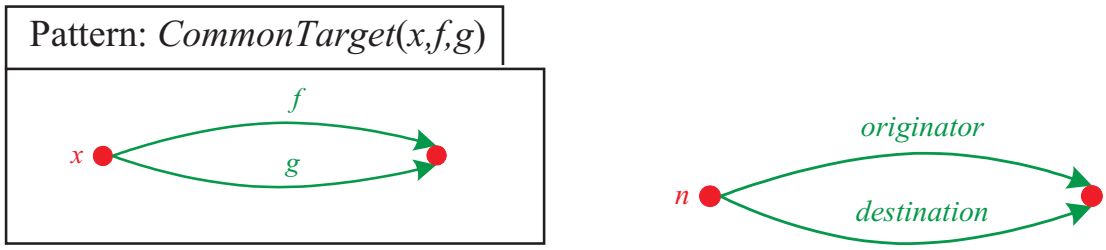

Figure 5: Common target

Pattern 3: Common target

The application of two functions to a value gives the same result. This pattern is shown in Figure 5. The pattern name is *CommonTarget* and the parameters are the value x and the functions f and g. The pattern constraint asserts that the functional images of x under both f and g yield the same element, which is distinct from x.

The diagram on the right hand side of Figure 5 shows an instantiation of the pattern that comes from a medical information specification. It is $CommonTarget(n, originator, destination)$ and asserts that the originator and destination of (note) n must be common; for example, a doctor writes a note to herself.

In OCL this pattern can be stated as $inv : x.f = x.g$ *and* $x <> x.f$ *and* $x <> x.g$, where f and g represent role names of two associations between class A and class B with multiplicity 1 at B's end.

Pattern $DisjointTargets$ was specified in terms of relations while pattern *CommonTarget* was specified in terms of functions. We could have specified variations of each pattern in terms of functions or relations, respectively. There are generalizations of these patterns that could involve, for example, the subset relation on the relational images.

4.3 Constraint Patterns in Operation Specifications

In this section we consider some patterns that frequently occur when specifying operations in terms of preconditions and postconditions. So the type of constraints the patterns provide are either a precondition or postcondition.

Pattern 4: Add element

This pattern is ubiquitous (Figure 6). It is common in many specifications when an element is to be added to a set or collection. The pattern name is $AddElement$ and the parameters are the set A and the element x. The pattern constraint has two parts. The first part is a precondition

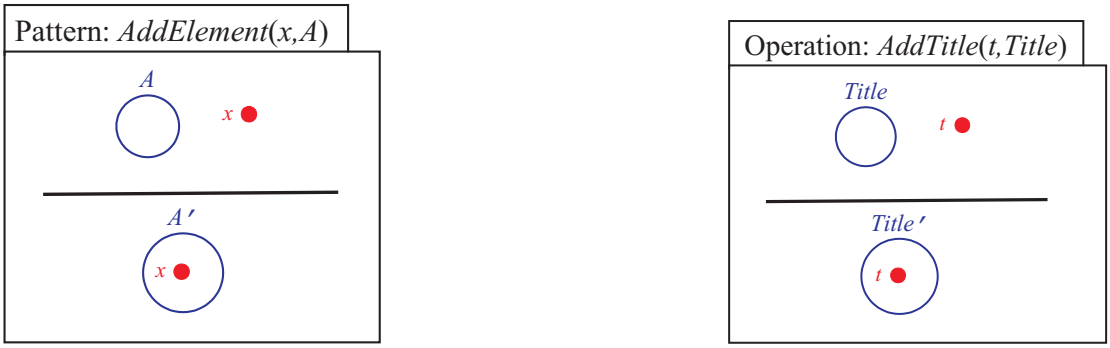

Figure 6: Add element

constraint (above the line) which states that the element x is not currently a member of the set A. The other part is a postcondition constraint (below the line) which states the effect of applying the operation $AddElement$ namely that x becomes an element of A.

An instantiation of this pattern is shown on the right of Figure 6. It is $AddElement(t, Title)$ and specifies an operation that adds a new title t to an existing set of titles in the context of a library specification. Recall that the framework for the diagrammatic notation assumes that "the rest remains unchanged" so that no existing titles are removed and no title other than t is added.

We can extend this pattern to include type information T for the elements. This is shown in Figure 7. In this case the type of the element x and the elements of set A are made explicit. Type information could, of course, be added to any pattern, such as the patterns considered earlier.

In OCL the precondition and postcondition constraints of the *Add Element* operation can be stated as $pre: A \rightarrow excludes(x)$ and $post: A \rightarrow includes(x)$ respectively. Here pre : indicates that the type of constraint is a precondition and $post$: indicates that the constraint is a post-condition. For the library example, if $catalog$ represents the set of titles for the library at a point in time, then the precondition takes the form $catalog \rightarrow excludes(t)$ and the postcondition takes the form $catalog \rightarrow includes(t)$. Note that the OCL postcondition states the minimal requirement with the implicit assumption that the rest of the set remains unchanged. An alternative way to write the postcondition is to explicitly state that the rest remains unchanged as $catalog = catalog@pre \rightarrow union(Set\{t\})$, where $catalog@pre$ denotes the set of titles at precondition time.

Pattern 5: Add subset

The *Add Element* pattern can be generalized to add a set of elements to a set which we name the *AddSubset* pattern. This pattern is shown in Figure 8. The parameters of the pattern are sets A and X. The precondition constraint states that A and X are disjoint sets. The postcondition con-

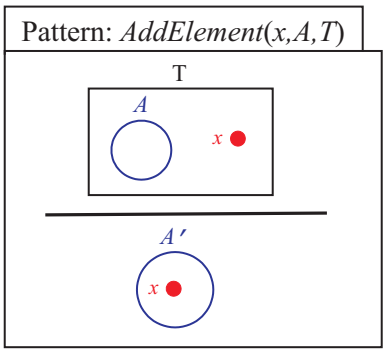

Figure 7: Add element

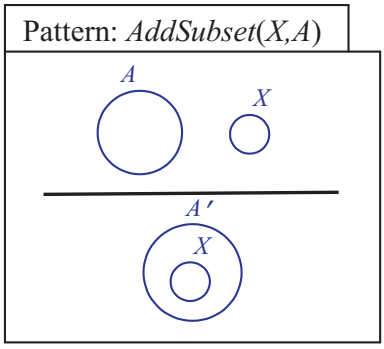

Figure 8: Add subset

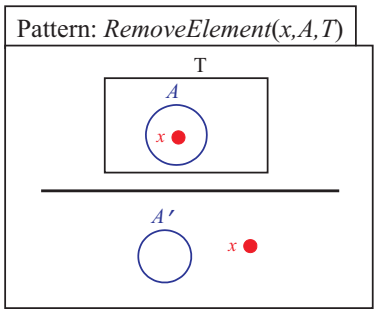

Figure 9: Remove element

straint states that X is a subset of A. The condition for applying the pattern is that the elements of both A and X are of the same type. Type information can also be included in the diagram.

Describing the add subset pattern in OCL is similar to the add element pattern. For the precondition we use the operation *excludesAll* instead of *excludes*, thus $pre: A \rightarrow excludesAll(X)$. For the postcondition we replace *includes* with *includesAll*, thus $post: A \rightarrow includesAll(X)$.

Pattern 6: Remove element

Removing an element from a set is also a very common pattern. This pattern is shown in Figure 9. The name of the pattern is *Remove Element* and the parameters include the element x, the set A and the type T. The pattern can be applied under the condition that x and the elements of A are of type T. The pattern constraint consists of the precondition which asserts that the element to be removed is already in the set, and the postcondition which states that x is no longer an element in the set A. The remove element pattern can stated in OCL as $pre: A \rightarrow includes(x)$ and $post: A \rightarrow excludes(x)$.

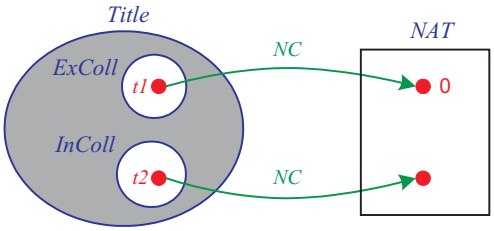

Figure 10: Combining patterns

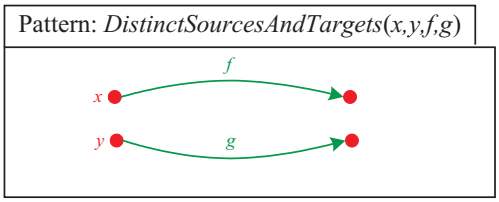

Figure 11: Distinct sources and targets

5 Combining Patterns

Visual specification patterns can be combined in a variety of ways to specify more complex constraints. The most obvious way of combining patterns is by simple conjunction. In the diagrammatic framework these could be represented by two separate diagrams and frequently this would be the most appropriate way of expressing the combination. However, there are more interesting ways of combining patterns.

Figure 10 shows a constraint on a library system that specifies that the number of copies (in the library's collection) of an ex-collection title is zero, while an in-collection title has a positive number of copies. The two subsets of the set *Title*, *Ex-Coll* and *In-Coll* are disjoint and partition the set *Title*. *NAT* represents the set of natural numbers and *NC* is a function that takes a title and delivers the number of copies of that title in the library's collection. Visually, we can see that the number of copies of an *Ex-Coll* title *t1* is zero, as *t1* is mapped to the element 0 by the arrow *NC* and the number of copies of an *In-Coll* title *t2* is positive, because the arrow *NC* maps *t2* to a natural number that is not zero, as it is distinct from zero, and is hence positive.

Figure 11 shows a variation on the *Disjoint Targets* pattern, called *DistinctSourcesAndTargets*. A combination of the binary partition pattern (Figure 3), $BinaryPartition(Title, ExColl, InColl)$, and the distinct targets pattern, $DistinctSourcesAndTargets(t1, t2, NC, NC)$, can be used to construct the constraint represented in Figure 10. However, combining visual patterns is non-trivial. In this example, the elements $t1$ and $t2$ need to be placed in the circles *ExColl* and *InColl*, respectively. Defining a framework to facilitate this is challenging and will be considered in further work. Finally, some post-application annotation is required in order to label one of the elements 0, although this could also be achieved by adapting *DistinctSourcesAndTargets* to include extra, optional, parameters to label the target elements.

6 Related Work

In the context of object-oriented modelling and component specifications, OCL is considered to be a very important formalism. However, developing concise and correct constraints in OCL is difficult. This problem has been addressed by numerous publications some of which use the idea of constraint patterns in order to capture recurring domain constraints and make it reusable.

Patterns for *constraints* in model-driven development were considered in [BHSS00], which

proposes a mechanism for connecting design patterns with OCL constraints; it enables the instantiation of OCL constraints automatically whenever a design pattern is instantiated. The notion of constraint patterns is further elaborated in [AT06, ABB+05], where a small number of simple constraint patterns is presented along with OCL templates.

The two publications [EM05, CGQ+06] introduce a larger number of constraint patterns. The patterns presented there originate from the data modelling domain. [WKB07] builds on these approaches by introducing composite patterns which allows users to negate patterns and to combine existing patterns using logical connectives such as implication. A category-theoretic language independent approach to pattern formalization and composition is presented in [BGL10].

Our contribution adopts the idea of constraint patterns to diagrammatic modelling using constraint diagrams and other diagrammatic notations. By mapping some of the OCL constraint patterns to constraint diagrams one gets the benefit of visualizing OCL constraints in the context of object-oriented modelling using UML.

7 Discussion and Future Work

We have identified some basic diagrammatic constraint patterns that occur frequently in simple case studies, with the intention of enabling the development of a tool, with access to a library of constraint patterns, that enables a user to instantiate these patterns within their modelling framework. The approach is flexible in general, allowing the use of the diagrams as both a specification and modelling language, or just allowing the use of the diagrams as a constraint language for placing constraints over some metamodel in the same manner as OCL is used, for example.

Of course, we have only presented some examples of patterns, and one has no means of deciding when one has "enough" patterns. However, as one means of measurement, we can create diagrammatic constraint patterns covering all of the examples of Ackermann [Ack05b]. However, as usual there is a trade-off between the benefits of utilising diagrammatic notations and textual notations, and attempting to understand the pro/cons of the choice of textual/diagrammatic notation for modelling constraints is one future avenue of interest. Attempting to classify constraints into preferential categories of textual, diagrammatic, or either would enable tools to offer choices of presentation of constraints, although of course this brings the disadvantage of requiring the use of more than one modelling notation. For example, some constraints, such as Ackermann's 2nd pattern, "Invariant for a property value (e.g. age $>$ 18)", appear to be be more naturally expressed in textual format, whereas constraints that can be formulated to make use of spatial relationships such as containment or exclusion (e.g. set membership or containment) benefit from the diagrammatic notation.

In terms of the approach to pattern development adopted, one can ask the question of how natural certain patterns are within the modelling language. Here, we took the stance of developing patterns with reference to existing diagrammatic case studies. As such, one can ask the question: are all of the constraint patterns that arise inherent in the models we are dealing with, or are some of them arising due to the modelling language itself? For example, properties like the uniqueness of some attribute would seem to be inherent, whilst the modelling language may tailor the type of model and constraint one constructs and so specific forms of constraints may

occur. We observe that there is an overlap between the OCL constraint patterns that Ackermann, who took an OCL-oriented developmental approach, identifies, and the diagrammatic constraint patterns presented here, but that they are not the same; some constraints stand out more in one language as opposed to the other (e.g. disjoint targets in the diagrams setting versus invariant attribute value for textual setting).

There are obvious extensions of the constraint patterns such as extend binary partition to n-ary partition, depicting this using ellipsis, and similarly for disjoint targets, for instance. Another area of future investigation is the development of alternative patterns, within the same language, for the same constraints. For example, using diagrams one can represent relations using labelled arrows, or by allowing sets to be labelled by the Cartesian product of two sets, thereby utilising a curve/label based representation. Although the use of arrows appears to be the most natural way of visualising functions or relations in this notation, when one considers the combination of constraints or constraint patterns, the choice of representation will play an important role in how well such patterns fit together.

The avenue of facilitating the combination of constraint patterns is likely to bring great benefits, extending the expressivity of the approach (enabling the specification of more complex constraints, such as nested constraints) in a natural manner. This approach could take several forms, but one could consider enabling the combination of two patterns that have something (like a set) in common, enabling the building of more complex constraints over the model. One could allow the matching of patterns within other patterns, such as asserting the uniqueness of an attribute after specifying some constraint that contains that attribute. In these cases, the point is that we are allowing the patterns to match not only over the model itself but also over the constraints already constructed. Further work also involves the development of a formal framework to facilitate the use of diagrammatic constraint patterns in specification; such a framework could be based on [BGL10].

Bibliography

[ABB+05] W. Ahrendt, T. Baar, B. Beckert, R. Bubel, M. Giese, R. Hähnle, W. Menzel, W. Mostowski, A. Roth, S. Schlager, P. H. Schmitt. The KeY tool. *Software and System Modeling* 4(1):32–54, 2005.

[Ack05a] J. Ackermann. Formal Description of OCL Specification Patterns for Behavioral Specification of Software Components. In Baar (ed.), *Proceedings of the MoDELS'05 Conference Workshop on Tool Support for OCL and Related Formalisms - Needs and Trends, Montego Bay, Jamaica, October 4, 2005*. Technical Report LGL-REPORT-2005-001, pp. 15–29. EPFL, 2005.

[Ack05b] J. Ackermann. Frequently Occurring Patterns in Behavioral Specification of Software Components. In *Component-Oriented Enterprise Applications. Proceedings of the Conference on Component-Oriented Enterprise Applications (COEA*. Pp. 41–56. 2005.

[AT06] J. Ackermann, K. Turowski. A Library of OCL Specification Patterns for Behavioral Specification of Software Components. *Lecture Notes in Computer Science* 4001:255–272, 2006.

[Baa05] T. Baar. OCL and Graph-Transformations - A Symbiotic Alliance to Alleviate the Frame Problem. In *Proc. MoDELS Satellite Events 2005*. Pp. 20–31. 2005.

[BGL08] P. Bottoni, E. Guerra, J. de Lara. Enforced generative patterns for the specification of the syntax and semantics of visual languages. *JVLC* 19(4):429–455, 2008.

[BGL10] P. Bottoni, E. Guerra, J. de Lara. A language-independent and formal approach to pattern-based modelling with support for composition and analysis. *Information and Software Technology* 52(8):821–844, 2010.

[BHSS00] T. Baar, R. Hähnle, T. Sattler, P. H. Schmitt. Entwurfgesteuerte Erzeugung von OCL-Constraints. *Softwaretechnik-Trends* 20(3), 2000.

[BKPT00] P. Bottoni, M. Koch, F. Parisi-Presicce, G. Taentzer. Consistency Checking and Visualization of OCL Constraints. In *Proc. UML 2000*. Lecture Notes in Computer Science 1939, pp. 294–308. Springer, 2000.

[CD00] J. Cheesman, J. Daniels. *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2000.

[CGQ+06] D. Costal, C. Gómez, A. Queralt, R. Raventós, E. Teniente. Facilitating the Definition of General Constraints in UML. In *MoDELS*. Lecture Notes in Computer Science 4199, pp. 260–274. Springer, 2006.

[DW98] D. F. D'Souza, A. C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, Reading, 1998.

[EM05] L. N. Elita Miliauskaite. Representation of Integrity Constraints in Conceptual Models. *Information Technology and Control* 34:355–365, 2005.

[FFH05] A. Fish, J. Flower, J. Howse. The Semantics of Augmented Constraint Diagrams. *JVLC* 16:541–573, 2005.

[Gur01] C. Gurr. Aligning syntax and semantics in formalisations of visual languages. In *In* Proceedings of IEEE Symposia on Human-Centric Computing Languages and Environments. Pp. 60–61. IEEE Computer Society Press, 2001.

[Hal90] A. Hall. Seven Myths of Formal Methods. *IEEE Software* 7(5):11–19, 1990.

[HJR02] R. Hähnle, K. Johannisson, A. Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In *Fundamental Approaches to Software Engineering, 5th International Conference FASE*. Pp. 233–248. Grenoble, June 2002.

[HS05] J. Howse, S. Schuman. Precise visual modeling: A case-study. *Software and System Modeling* 4(3):310–325, 2005.

[Ken97] S. Kent. Constraint diagrams: visualizing invariants in object-oriented models. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. Pp. 327–341. ACM, New York, NY, USA, 1997.

[LDF04] Y. Ledru, S. Dupuy, H. Fadil. Towards Computer-Aided Design of OCL Constraints. In *Proceedings of CAISE'04 Workshops Vol. 1 -EMMSAD'04: Evaluating Modeling Methods for Systems Analysis and Design*. Pp. 329–338. Riga, June 2004.

[SGM02] C. Szyperski, D. Gruntz, S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, NY, second edition edition, 2002.

[SPB90] S. Schuman, D. Pitt, P. Byers. Object-oriented process specification. In *Rattray (ed.)* Specification and Verification of Concurrent Systems*, Proc. FACS Workshop, Springer*. Pp. 21–70. 1990.

[WK03] J. Warmer, A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, Reading, MA, 2003.

[WKB07] M. Wahler, J. Koehler, A. D. Brucker. Model-Driven Constraint Engineering. *Electronic Communications of the EASST* 5, 2007.
http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/44/70