A Visual Framework for Modelling with Heterogeneous Notations

Jean Flower, John Howse, John Taylor
School of Computing & Mathematical Sciences
University of Brighton, Brighton, UK
{J.A.Flower, John.Howse, John.Taylor}@brighton.ac.uk

Stuart Kent
Computing Laboratory
University of Kent, Canterbury, UK
S.J.H.Kent@ukc.ac.uk

Abstract

There is a range of modelling notations, both textual and diagrammatic, whose semantics are based on first-order predicate logic. This paper presents a visual framework for organizing models of systems which allows a mixture of notations, diagrammatic or text-based, to be used. The framework is based on the use of templates for connective operations which can be nested and sometimes flattened. It is modular and can be used to structure the constraint space of the system, making it scalable with appropriate tool support. It is also flexible and extensible: users can choose which notations to use, mix them and add new notations or templates.

The goal of this work is to provide more intuitive and expressive languages and frameworks to support the construction and presentation of rich and precise models.

Keywords Visual formalisms, software specification, heterogeneous framework.

1 Introduction

There is a range of notations available for the modelling of software systems whose semantics are based on first-order predicate logic (FOPL). Text-based examples include FOPL itself, or structured variants such as Z [19] and VDM [13], and the *Object Constraint Language* (OCL) [20], which is part of the *Unified Modelling Language* (UML) [16]. Diagrammatic examples include *constraint diagrams* [7] and UML *class* and *state diagrams*. A heterogeneous framework in which these and other notations can be used together might be of benefit to modellers. In this paper we present such a framework, discussing the advantages and disadvantages of different approaches.

Part of the rationale in the development of both OCL and constraint diagrams was to construct a developer-friendly notation for expressing constraints in object-oriented modelling, as an alternative to traditional mathematical syntax. Notation is notoriously a matter of taste. Having a range of notations available enables us to choose which is the most appropriate approach for our needs or our tastes. Some expressions, such as those which require navigation between sets and statements about set inclusion and disjointness are probably shown better diagrammatically than textually; other statements, such as those involving numbers, are frequently better expressed textually.

A further problem when modelling industrial-sized systems is scalability; the number and complexity of constraints can be overwhelming. The framework developed in this paper is modular and can be used to structure the constraint space of the system. It is also flexible and extensible. Users can choose which notations to use and can mix them; they can choose the ways in which the notations are combined and they can add their own notations. All the formal notations in the family should be based upon FOPL, and expressions are built from atomic expressions using connectives. The framework is flexible enough to deal with informal notations such as natural language or rich pictures and also examples such as UML *object diagrams*. Expressions within the framework will have formal semantics if and only if all their components are formally defined.

Diagrammatic reasoning can take many forms as can be evidence by glancing at [1] or [9]. In [3], Barwise and Etchemendy argue that reasoning is heterogeneous in nature. They applaud the recent resurgence of interest in "nonlinguistic" representations in reasoning but strike a note of caution about the potential of nonlinguistic representations. Just as it is unreasonable to suggest that first-order logic is a *universal* representational language, it is also unreasonable to strive for a universal nonlinguistic representational language. They suggest that the search for *any* universal scheme of representation, text-based or diagrammatic, is a mistake and that reasoning is inescapably heterogeneous in nature. We fully concur with these sentiments.

The framework presented in this paper has developed from earlier work. In [14], we showed how OCL and constraint diagrams can be used effectively to-

gether and in [15] we made the first suggestions for a framework, *constraint trees*, for mixing different notations; constraint trees form a subset of the framework we present in this paper.

In the next section, we introduce the concept of a *template*. Our framework for heterogeneous notations is built from templates, and users can add their own templates. First we consider binary, commutative and associative connective operations such as conjunction and disjunction. They can be extended to n-ary operations because they are associative. Later, we consider other operations which may not be commutative (e.g. implication) or binary (e.g. negation). One use of unary templates for predicates and quantifiers is outlined in §2.6. In §3 we show how templates can combine in a nested expression and discuss flattening nested expressions. In §4 we formalise the notation. We conclude with discussions on tool support and related work.

2 Templates

An expression is either *atomic* or made up of components, using some connective operation. A *view* of an expression is a concrete (diagrammatic or text-based) representation of an expression. A view can be built up out of views of its components. The component views are combined using a *template*. We will consider the templates *tree*, *chain*, *box*, and *partition*, see figures 1 and 3.

To create a view of an expression, use a template for each application of a connective operation. The framework is flexible and extensible, because users can introduce their own templates or adapt existing ones. The templates we introduce here serve only as examples.

2.1 Binary commutative templates

Binary operations are shown using binary templates. An outer rectangle contains two empty inner rectangles to hold views of the components. The space between the bounding rectangle and the inner rectangles is filled differently for different templates. Commutative operations allow templates that don't enforce an ordering on the inner rectangles. The component parts can be read in any order. The simplest binary commutative template is *box*, which simply has two rectangles drawn inside a bounding box. A line can be drawn between the inner rectangles to give a different template: *chain*.

A third template uses a new rectangle (smaller than the others) to represent the operation which brings together the components. Lines link the view of the operation to the inner rectangles. This template is called

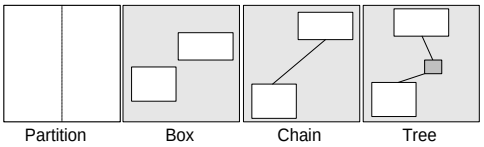

Fig. 1: Four binary templates

tree. The fourth possibility shown in figure 1 makes most efficient use of space on the diagram. The two inner rectangles occupy the whole of the bounding rectangle. One line has been used to *partition* the bounding box into parts, ready to hold views of the components.

The *partition* template is inspired by the use of dashed lines to indicate orthogonal behaviour in state diagrams in UML, which originated in Harel's statecharts [11]. The *chain* syntax draws upon our work on reasoning with spider diagrams [12], which itself builds upon the work of Shin [18] and Peirce [17]. The *tree* template develops from the idea of a constraint tree used for combining OCL and constraint diagrams [15].

2.2 n-ary commutative templates

If a binary operation is associative, then the repeated application of that operation can be constructed without explicit reference to the pairs of components which are combined first. Figure 2 shows how a binary commutative associative template can become a ternary template. In this way, the binary templates shown in

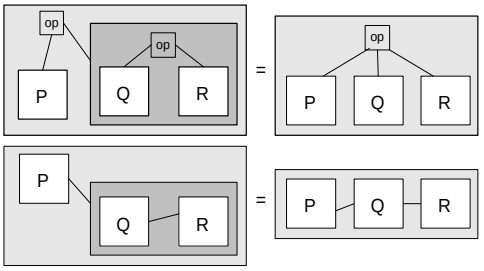

Fig. 2: The associative law

figure 1 become the ternary templates seen in figure 3 and, of course, the process can be extended to produce n-ary templates.

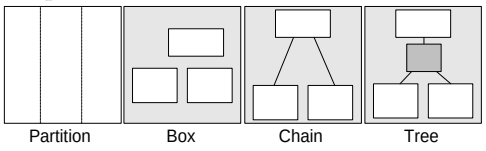

Fig. 3: Ternary templates

2.3 Use of operation annotations

For diagrams representing expressions involving different operations, it is useful to be able to annotate the

diagram with operation labels or signifiers. Figure 4 shows some annotated binary templates.

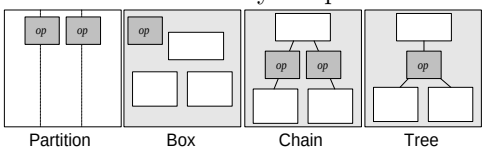

Fig. 4: Annotated templates

Sometimes we can use diagrammatic notations instead of textual annotations; examples of this are found in the next two subsections.

2.4 Unary templates

A unary template needs only a bounding box and a single inner box. Unary templates can be thought of as frames, or wrappers, of an inner view. The *not* operation is unary, and two possible templates are shown in figure 5, one in which the template is annotated with the label ¬ and the other in which the negation is indicated diagrammatically as a bar above the inner box. Probably the only unary operation we will require is

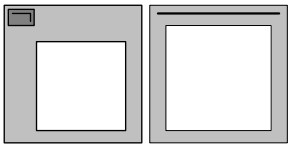

Fig. 5: Templates for *not*

negation. However, in §2.6, we show how a unary template can be used to express quantification.

2.5 Non-commutative templates

A non-commutative operation can be shown using a template which imposes an ordering on its inner boxes. Two possible templates for implication are shown in figure 6. In one the two boxes are linked with an arrow resembling an "implies" sign (⇒), while in the other the two boxes are linked with an arrow which is annotated with the label ⇒. Of course, we could have annotated the arrow with the word *implies* rather than the symbol. An *implies* template can only be binary as

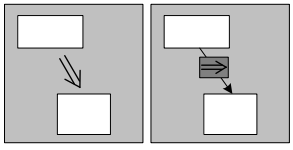

Fig. 6: Templates for implication

the operation is not associative. Another similar template could be developed for *if then else* statements. See §6 for an example from [4] for such a conditional construction.

2.6 Predicates and quantifiers

Quantifiers are easily represented by unary templates to represent the quantifier and variable name. The inner box of the template should then be filled with a predicate. The most obvious template is employs a simple text-based label as in figure 7 although one could envisage representations where universal and existential quantification are distinguished diagrammatically.

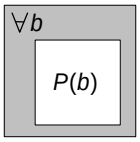

Fig. 7: A unary quantification template

The scope of a quantifier is represented directly by the bounding box. For example, figure 8 represents the proposition $\forall a\ \exists b \bullet P(a,b) \wedge Q(b)$ in the left-hand diagram and the predicate $\forall a \bullet P(a,b) \wedge (\exists b \bullet Q(b))$ in the right-hand diagram.

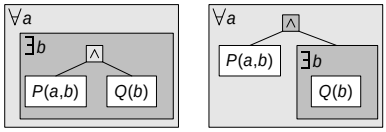

Fig. 8: The scope of quantifiers

3 Nesting templates

The nesting of templates allows for compound expressions to be built up which can contain different connectives. Figure 9 shows how different templates can combine in a nested expression. The most flexible notation is obtained by allowing the use of any template for any instance of any operator. A more rigid notation is obtained by insisting on a consistent use of a unique template for each connective. A *schema* could be chosen which uses the tree template for conjunction and the box template for disjunction. If a schema assigns unique templates for each operation, then annotations can be omitted and every diagram has well-defined semantics. On the other hand, a schema which duplicates the use of a template then requires annotation for those

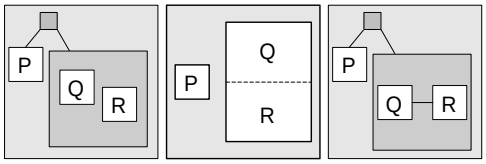

Fig. 9: Nested templates

occurrences of the template to specify which operation is being represented.

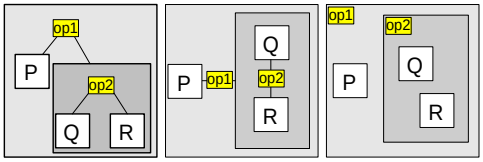

Fig. 10: Annotated nested templates

3.1 Modularity and scalability

In a modelling situation, we may wish to use notation such as natural language, without clear semantics. A mix of formal and informal statements allows for vague statements to be made, elaborated upon, and made precise at a later time. The semantics of nested expressions can be built up modularly. The semantics of a compound expression exists if and only if we can give semantics to each component part.

The modularity of the framework allows for scalable models. With appropriate tool support, see §5, we can zoom in and out of the model, exploding and collapsing views, expressing the whole model but concentrating only on manageable sized chunks of the system at any time; these chunks can range from an overview of the whole model down to very detailed views of the smallest components.

3.2 Flattening nested notations

Templates are required to have a bounding box in order to permit unambiguous nesting, but in some cases diagrams may be simplified by removing some of the bounding boxes. We call this process *flattening*. The associative law illustrated in Figure 2 is an example of flattening, but more interesting examples arise when more than one operation appears in the represented expression. The existing notations of spider diagrams [12] and constraint trees citekent:trees both employ flattening as illustrated respectively in Figure 11 (representing $(P \vee Q) \wedge (R \vee S \vee T)$) and Figure 12 (representing $P \wedge (Q \vee R)$).

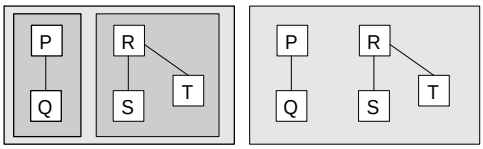

Fig. 11: Flattening in spider diagrams

Flattening is possible in some cases where each operation is associated with a particular template style. In the spider diagram notation, disjunction is always represented by chain templates and conjunction is always represented by box templates. Similarly, in the constraint tree notation both disjunction and conjunction are represented by tree templates. We call such an association of operations to template styles in this way a (*template*) *schema* (see §4 for the formal definition). Not all schemas admit flattening. For example, the 'box-box' schema that represents both conjunction and disjunction as box templates does not permit flattening; see Figure 13.

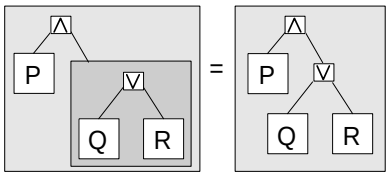

Fig. 12: Flattening in constraint trees

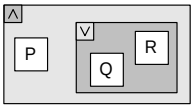

Fig. 13: A schema that does not admit flattening

We regard the chain and tree templates as being *connected* and the box template as being *disconnected*. Since connected templates are, in a visual sense, more 'tightly bound' than disconnected templates, a schema that assigns one operation to a connected template and one to a disconnected template has an implicit operator hierarchy that permits flattening. This is the diagrammatic counterpart of the operator hierarchy in algebra where multiplication binds more tightly than addition allowing the convention that $xy + z$ can be unambiguously interpreted as $(x \times y) + z$ rather than $x \times (y + z)$.

Flattening simplifies diagrams, but there is a cost. For those diagrams that employ a schema that assigns one operation to a connected template and one to a disconnected template, flattening imposes a normal form on the notation. For example, flattened spider diagrams (illustrated in the right-hand diagram of Figure 11) represent expressions in conjunctive normal form where disjunction binds more tightly than conjunction. Reversing the assignment (conjunction to chain templates and disjunction to box templates) produces diagrams that represent expressions in disjunctive normal form.

The constraint tree notation that employs the 'tree-tree' schema that assigns both disjunction and conjunction to tree templates admits flattened diagrams without a unique interpretation. Figure 14 illustrates such a diagram which has interpretations that represent $P \wedge Q \wedge (R \vee S)$ and $(P \wedge Q) \vee R \vee S$. This ambiguity can be resolved by highlighting one operator node as

the 'root' of the tree. The constraint tree can then be interpreted recursively with operator nodes connected to the root node becoming the root nodes of subtrees.

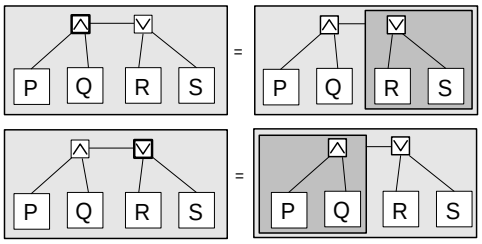

Fig. 14: The ambiguity of constraint trees

Another way to resolve the ambiguity is to present the tree with 'outermost' connectives higher on the page. The tree is then read in a top-down manner, as illustrated in Figure 15. In either case, identifying the topmost or root connective is sufficient to deduce the semantics of the whole tree. We could flat-

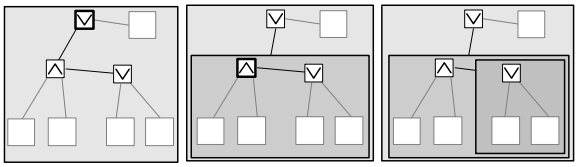

Fig. 15: Evaluation of a constraint tree from the root

ten the unary quantification template, replacing the quantification annotation $\forall b$ with a combined annotations $\forall b, \wedge$ on disconnected templates or annotating the shaded region inside the bounding box with $\forall b$ for connected templates (see figure 16).

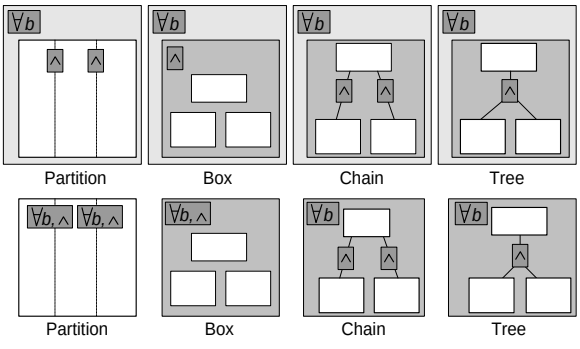

Fig. 16: A flattened quantification template

4 Formalising templates using nesting with schemas

A logical expression is either atomic or is the result of applying an *operation* to other expressions. Examples of operations are disjunction, conjunction and negation. Let E be the set of expressions and O be the set of operations. Figure 17 gives a UML class diagram for expressions.

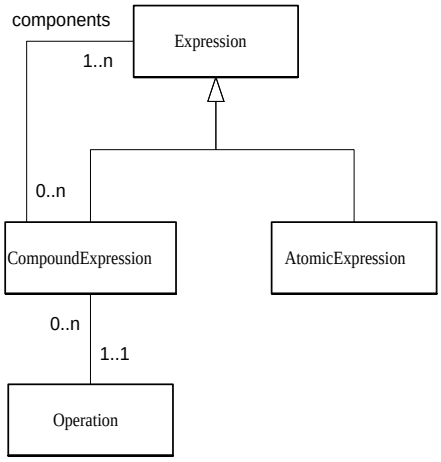

Fig. 17: UML class diagram for expressions

A *view* of an expression is a diagram contained in a bounding box which represents an expression. We assume that all atomic expressions have a concrete representation which can be placed into a box, giving an atomic view of the atomic expression. Views of compound expressions will be built up out of atomic views using templates. Let V be the set of views. Figure 18 gives a UML class diagram for views.

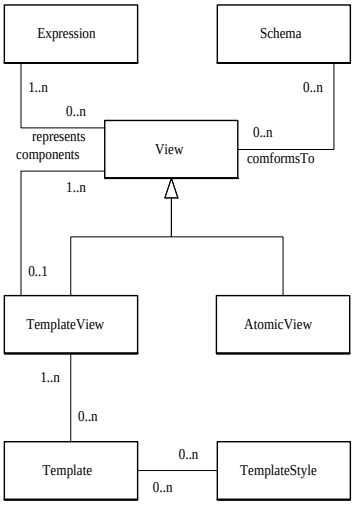

Fig. 18: UML class diagram for views

An n-ary *template* is a diagram consisting of one bounding box containing everything else, and n disjoint, empty, inner boxes. The space between the outer box and the inner boxes may contain shading, lines, annotation, or other diagrammatic components. An n-ary template is also referred to as a template of *degree* n. Let T be the set of templates. A template is capable

of being displayed by a tool. We could, for example, choose a template which is presented on screen with empty inner boxes, and drag and drop views of expressions into the inner boxes to turn the template into a view. Thus an n-ary template defines a function:

$$t \in T \Rightarrow t : V^n \to V$$

If an n-ary template has $n > 1$, and an inner box is filled with a view, then we have a new template of degree $n - 1$. The operation $\wedge : (P, Q) \to P \wedge Q$ can be represented by a binary box template, and the operation $op : (P, Q) \to P \wedge Q \wedge R$ can also be represented by a binary box template (left diagram of figure 19), but would be better expressed using a partially filled ternary template (right diagram in figure 19).

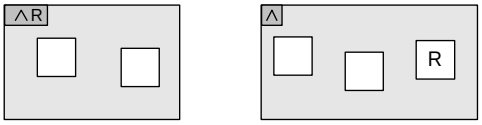

Fig. 19: A partially-filled template

A *template style* is a rule which determines what kinds of diagrammatic structure is present between the bounding box and the inner boxes of a template. One example of a template style, called *box*, requires that the only feature of a template between the bounding and inner boxes is shading. A template is said to *conform to* a template style. A second example of a template style is *chain*, which says that the inner boxes are joined together in a chain using line segments in the bounding box. Some template styles can apply to only templates of certain degree, whereas others can apply to templates of any degree. Let TS be the set of template styles. Say that a template *conforms to* a template style if the template structure satisfies the template style rule. If template $t \in T$ conforms to template style $ts \in TS$, we write $t \sim ts$. In principle, a template could conform to more than one template style, but the four styles described in this paper are mutually exclusive.

A view and a template are concrete diagram concepts. One expression (abstract syntactic) can have many (concrete) views and one template style (abstract) can have many (concrete) templates conforming to that style.

A *schema* assigns a template style to every operation in a logical system. For example, schema s could assign the box template style to the disjunction operation, and the chain template style to the conjunction operation. Let the set of schemas be S. Each schema s defines a function:

$$s \in S \Rightarrow s : O \to TS$$

Figure 20 gives a UML class diagram for schema. If

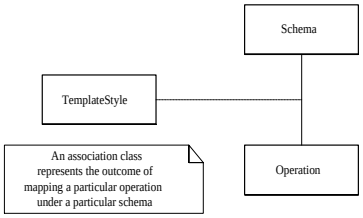

Fig. 20: UML class diagram for schema

an expression is non-atomic, then the view must be composed of a template, with views of the component parts drawn within the inner boxes of the template. Take $e \in E$ such that $e = op(c_1, .., c_n)$ and views, v_e of e, and v_i of each component c_i. Then there exists a template t such that

$$v_e = t(v_1, .., v_n).$$

A view is said to *conform to a schema* s if each instance

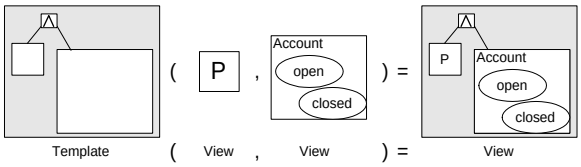

Fig. 21: Combining views with a template

of an operation within the expression is represented by a template which conforms to the relevant template style. Take $e \in E$ such that $e = op(c_1, .., c_n)$ and views v_e of e, and v_i of each component c_i. Assume these views conform to schema s. Then there exists a template t such that

$$(t \sim s(op)) \wedge (v_e = t(v_1, .., v_n)).$$

5 Tool Support

Using visual notations in modelling requires tool support. As soon as one wishes to maintain the diagrams, use them in formal documentation, check their correctness, reason with them, manipulate them, and so on, then tool support is essential. A specialized constraint diagram editor, the CD Editor [8], has already been developed.

The main requirement of a tool to support our framework is zooming, that is, changing the depth of a view by exploding or collapsing it. Of course, for any non-trivial system it would be impossible to fit the whole model on a single screen. A means, therefore, has to be provided for zooming in and out of views.

The tool should support the creation of new templates and the adaptation of existing templates. It

should "know" the framework's notations, diagrammatic and textual, and be able to support manipulation of these notations, and the results of any manipulation of the model must be presented in the notations that the modeller is using.

Each notation should have a set of logical reasoning rules, preferably complete, so that modellers can reason about the system within the notations they are using. The tool should also be able to evaluate any statement represented by any of the formal notations within the framework. It should also be possible to add new notations to the framework.

Another longer term possibility for tool support is notation interchange. It would be desirable for a tool to interchange the notation used to present the contents of a view, for example converting a constraint diagram into textual notation, and vice-versa. Work is already underway on this: we can generate Euler diagrams [5], the basic notation on which constraint diagrams are built, from abstract (textual) mathematical specifications [6]. However, the main challenge here is the layout of diagrams when these are created from text.

6 Related Work

The idea for heterogeneous systems is, of course, not new. For example, Barwise and Etchemendy have implemented a heterogeneous logical system, called Hyperproof [2], that mixes diagrammatic and textual notations and allows reasoning to be performed using the the appropriate notation. See [1] for other examples of heterogeneous systems.

Bottoni et al [4] use UML collaborations to give a visualization of OCL. They use a structuring method that would fit nicely into our framework. For example figure 22 is their visualization of an *if then else* statement. They criticize the approach taken in [14] of mixing OCL and constraint diagrams as "suffering the typical difficulties of parsing together diagrams and text". We claim that the framework developed in this paper overcomes these difficulties. The criticism is also counter to Barwise and Etchemendy's view that reasoning (and we would say modelling) is heterogeneous in nature.

Peirce's *alpha diagrams* [17] (but see also [10] for a more recent interpretation) use labels for propositions, simple closed curves surrounding diagrammatic elements for negation, juxtaposition for conjunction and nothing else. The notation is nested and fits neatly into our framework. The alpha diagram in figure 23 represents $\neg(\neg P \wedge \neg Q)$ and hence, rather non-intuitively, $P \vee Q$. The nested structure of the notation is clearly seen. Peirce's work on diagrammatic representations of

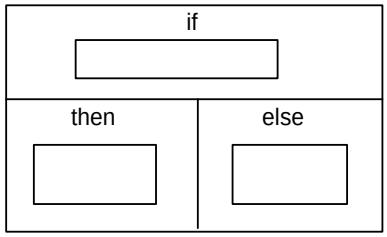

Fig. 22: A visualization of *if then else*

logic has been very influential on the recent resurgence of interest in diagrammatic reasoning.

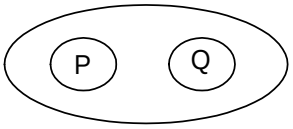

Fig. 23: A Peirce alpha diagram

7 Conclusion and Further work

There are a number of ways in which the framework and its components can be adapted and extended. For example, the tree template can be modified to reduce duplication of propositions by merging instances of a statement. Allowing merging of views for identical ex-

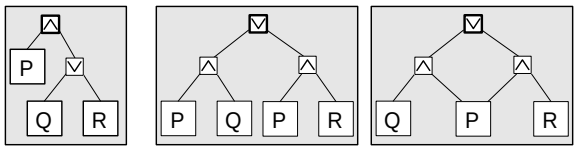

Fig. 24: The distributive law with merged propositions

pressions would change a constraint tree into a *constraint graph*. Work is in progress to clarify the notation required for constraint graphs. It is not enough, for example, to specify a root node of a constraint graph to deduce the semantics.

In this paper we have developed the idea of a visual framework for organizing models of systems which allows a mixture of notations, diagrammatic or text-based, to be used. The framework is based on the use of templates for connective operations which can be nested and sometimes flattened. It is modular and can be used to structure the constraint space of the system, making it scalable with the appropriate tool support. It is also flexible and extensible: users can choose which notations to use, mix them and add new notations or templates.

The goal of this work is to provide more intuitive and expressive languages and frameworks to support the construction and presentation of rich and precise models.

Acknowledgements This research was partially supported by UK EPSRC grants GR/R63509 and GR/R63516.

References

[1] G. Allwein and J. Barwise. *Logical Reasoning with Diagrams*. OUP, 1996.

[2] J. Barwise and J. Etchemendy. *Hyperproof*. CLSI, Stanford, 1994.

[3] J. Barwise and J. Etchemendy. Heterogeneous logic. In J. Glasgow, N. H. Narayan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning*, pages 211–234. MIT Press, 1995.

[4] P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of ocl using collaborations. In *Proceedings of UML01*, 2001.

[5] L. Euler. Lettres a une princesse d'allemagne. Letters Vol 2, No. 102-108, 1761.

[6] J. Flower and J. Howse. Generating Euler diagrams. In *Proceedings of Diagrams 2002*. Springer-Verlag, 2002.

[7] J. Gil, J. Howse, and S. Kent. Towards a formalization of constraint diagrams. In *Proc Symp on Human-Centric Computing*. IEEE Press, Sept 2001.

[8] J. Gil and Y. Sorkin. The constraint diagrams editor. Available at www.geocities.com/ysorkin/cdeditor/.

[9] J. Glasgow, N. Hari Narayanan, and B. Chandrasekaran, editors. *Diagrammatic Reasoning*. MIT Press.

[10] E. Hammer. Peircian graphs for propositional logic. In G. Allwein and J. Barwise, editors, *Logical Reasoning with Diagrams*, pages 129–147. OUP, 1996.

[11] D. Harel. On visual formalisms. In J. Glasgow, N. H. Narayan, and B. Chandrasekaran, editors, *Diagrammatic Reasoning*, pages 235–271. MIT Press, 1995.

[12] J. Howse, F. Molina, and J. Taylor. On the completeness and expressiveness of spider diagram systems. In *Proceedings of Diagrams 2000*, pages 26–41. Springer-Verlag, 2000.

[13] C. Jones. *Systematic Software Development using VDM*. Prentice Hall, 1990.

[14] S. Kent and J. Howse. Mixing visual and textual constraint languages. In *Proceedings of UML99*, 1999.

[15] S. Kent, J. Howse, and S. Gaito. Constraint trees. In A. Clark and J. Warmer, editors, *Advances in Object Modelling with OCL*. Springer Verlag, to appear, 2002.

[16] OMG. UML specification, version 1.3. Available from www.omg.org.

[17] C. Peirce. *Collected Papers*, volume 4. Harvard Univ. Press, 1933.

[18] S.-J. Shin. *The Logical Status of Diagrams*. Cambridge University Press, 1994.

[19] J.M. Spivey. *The Z Notation*. Prentice Hall, 1989.

[20] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, 1998.