Electronic Notes in Theoretical Computer Science 72 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 10 pages

Modelling with Heterogeneous Notations

Jean Flower, John Howse and John Taylor

School of Computing & Mathematical Sciences
University of Brighton, Brighton, UK
{J.A.Flower, John.Howse, John.Taylor}@brighton.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 [14] and VDM [8], and the *Object Constraint Language* (OCL) [15], which is part of the *Unified Modelling Language* (UML) [11]. Diagrammatic examples include *constraint diagrams* [4] 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 [5]. In [2], 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.

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.

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*.

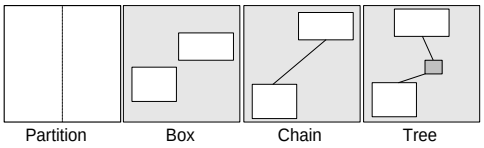

Fig. 1. Four binary templates

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 *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 [6]. The *chain* syntax draws upon our work on reasoning with spider diagrams [7], which itself builds upon the work of Shin [13] and Peirce [12]. The *tree* template develops from the idea of a constraint tree used for combining OCL and constraint diagrams [10].

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

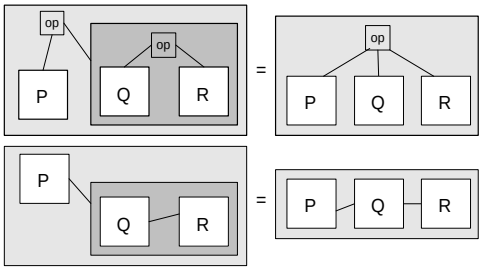

Fig. 2. The associative law

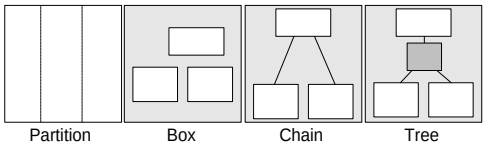

Fig. 3. Ternary templates

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.

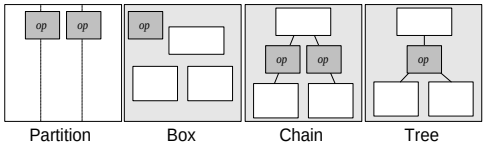

Fig. 4. Annotated templates

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

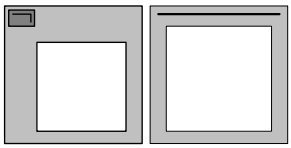

Fig. 5. Templates for *not*

the only unary operation we will require is 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

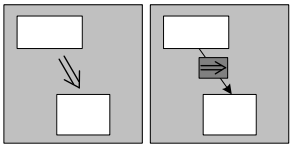

Fig. 6. Templates for implication

only be binary as the operation is not associative. Another similar template could be developed for *if then else* statements. See §6 for an example from [3] 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 employs a simple text-based label as in figure 7 although one could envisage representations where universal and existential quantification are distinguished diagrammatically.

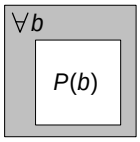

Fig. 7. A unary quantification template

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.

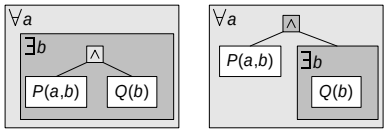

Fig. 8. The scope of quantifiers

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

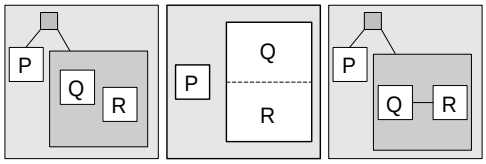

Fig. 9. Nested templates

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

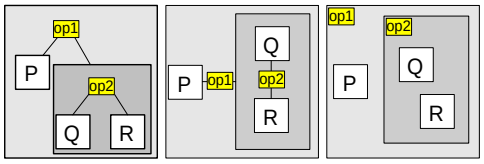

Fig. 10. Annotated nested templates

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

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 [7] and constraint trees [10] both employ flattening as illustrated respectively in Figure 11 (representing $(P \vee Q) \wedge (R \vee S \vee T)$) and Figure 12 (representing $P \wedge (Q \vee R)$).

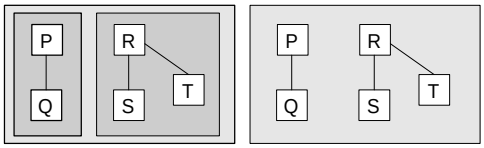

Fig. 11. Flattening in spider diagrams

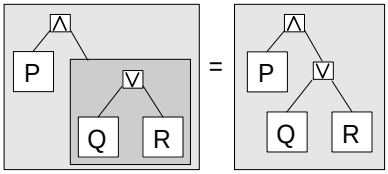

Fig. 12. Flattening in constraint trees

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.

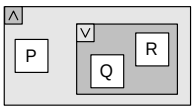

Fig. 13. A schema that does not admit flattening

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.

4 Formalisation

An n-ary *template* is a diagram consisting of n disjoint, empty boxes within a bounding box. 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 *view* of a logical expression is a representation of that expression in some concrete notation, which can be diagrammatic or textual, and is contained in a bounding box. A *template view* is a diagram in which each inner box of a template contains a view and the template contains a representation of a logical operation or quantifier. These views can be *atomic views* or template views. An *atomic view* does not contain any other view. Let V be the set of views, AV the set of atomic views and TV the set of template views. Then $V = AV \cup TV$ and $AV \cap TV = \emptyset$. Further, let Op be the set of operations, which can include quantifiers. Then $TV = Op \times V^n$. Note that an atomic view represents a logical expression which can include logical connectives and quantifiers; it is not necessarily an "atomic" expression.

Views and Templates are concrete concepts. An (abstract) logical expression can have many concrete views in many different notations. The *style* of a template determines what kinds of diagrammatic structure is present between the bounding box and the inner boxes of a template. 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 $t \sim ts$. We have considered four template styles in this paper: *box*, *chain*, *partition* and *tree*, but there are many other possibilities and users can develop their own styles. The differences between the styles is purely a matter of concrete notation and therefore of taste. However, some logical expressions cannot be expressed in some styles.

4.1 Semantics

Each view represents a logical expression. We define a semantic function Ψ that interprets each view as a logical expression in FOPL. Let $tv \in TV$. Then $tv = (op, v_1, ..., v_n)$ and

$$\Psi(tv) = \Psi(op)(\Psi(v_1), ..., \Psi(v_n)).$$

4.2 Flattening

The flattening processes described in §3.2 can be formalised as transformation rules. For example, associative binary operations can be extended to n-ary operations as described earlier. This gives rise to the following transformation rule:

The associative operation flattening rule. Let op be an associative operation. Let $tv = (op, v_1, ..., v_n)$ and $tv' = (op, tv_1, v_n)$ be template views where $tv_1 = (op, v_1, ..., v_{n-1})$. Then tv' can be replaced by tv.

Theorem 4.1 *The associative operation flattening rule is valid*

The proof is obtained by showing that $\Psi(tv) = \Psi(tv')$.

5 Conclusion

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 grant GR/R63516.

References

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

[2] 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.

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

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

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

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

[7] 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.

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

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

[10] 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.

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

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

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

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

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