Precise visual modeling: A case-study

John Howse, Steve Schuman

University of Brighton, UK
e-mail: {John.Howse,S.A.Schuman}@bton.ac.uk

Received: 19 December 2003/ Accepted: 23 August 2004
Published online: 20 April 2005 – © Springer-Verlag 2005

Abstract. We develop an abstract model for our case-study: software to support a “video rental service.” This illustrates how a visual formalism, constraint diagrams, may be used in order to specify such systems precisely.

Keywords: Constraint diagrams – Formal methods

1 Introduction

The aim of this paper is to illustrate a visual formalism for precisely specifying an information system. We adopt as our case-study a “video rental service” (which might be viewed as a variant on the classical “lending library” application). We shall approach this here in the spirit of a requirements engineering process [19]. Thus, our goal is to develop a high-level model (or more commonly, a succession of such models) serving to specify an abstract architecture for the required system – where its outcome must be understood and ultimately agreed by a client!

For that reason we use constraint diagrams, first proposed by [10] to visualize object-oriented invariants (in the context of UML [13]) and subsequently extended [5] to depict pre- and post-conditions as well. These build upon a long history of expressing logical or set-theoretic properties as diagrams: they generalize Euler circles [2] and Venn diagrams [21], which are once again rich research topics in their own right as the basis for visual formalisms or diagrammatic reasoning systems [7–9, 16]. Constraint diagrams are considerably more “expressive” than those underlying systems because they can specify functions or relations, whilst still retaining the elegance and intuitive appeal of a simple diagrammatic notation. Much progress has been made in formalizing constraint diagrams [3, 6]; the first diagrammatic reasoning system for (a cut-down version of) constraint diagrams has very recently been developed [20].

Such diagrams may also be used as a stepping-stone towards traditional mathematical specifications, e.g. in Z [17, 18]. The Appendix gives a corresponding object-oriented formal specification [14, 15] for our case-study. This starts with a short summary of that framework’s main conventions. Best known of these is “the rest stays unchanged” [12], which allows any post-condition to be expressed in a weaker (or minimal) form – significantly simplifying its usage in practice.

We will use exactly the same conventions to build up our diagrammatic specification in the body of this paper, encapsulating each separate level as a class of abstract objects. Hence, these two different formal specifications can be compared level-by-level. All basic conventions of constraint diagrams will be introduced by example.

2 An initial model

At the outset of any modeling process, one starts from a “blank sheet of paper.” Thus there is little choice but to follow an essentially bottom-up development strategy. Typically, we try to express earlier models as simply and abstractly as possible – deferring less important detail to later stages of formal refinement [1, 11] or indeed, until the final phase of system implementation.

We will develop our model incrementally, i.e. in successive stages. Figure 1 gives an overview of the initial

![Class-structure for simple rental service](image-url)
stage for this development: every level is depicted by a rectangle with a comment (//) to suggest its role; arrows indicate extension of classes and ● indicates composition.

2.1 Abstract membership: class VM [M, I]

We model such a membership as a finite initially-empty set of unique ‘identifiers’ (from given type M), each of which has some associated ‘information’ (from another given type I). This is introduced as class VM[M, I]. Its state-invariant along with its initial-condition (below the double line) are formally specified as follows:

\[
\text{VM} \quad \text{set } M \quad \text{Memb} \quad \text{Info} \quad I
\]

The invariant for class VM (above the double line) is a conjunction of two diagrams, separated by a semicolon: the first defines Memb to be an element of set M, i.e. a finite subset of given type M; the second, wherein this subset is shown as a circle, defines a relation Info that associates each element of Memb with some element of type I (so Info is a total function from Memb to I).

The shaded circle below the double line represents the empty set. Thus the initial-condition for objects of class VM might be read as “initially, there are no members.” Here, Memb’ denotes the value of set Memb when an object of the class is first instantiated; it follows that Info is the empty function in any initial state.

Closed curves or contours (usually in the form of circles or ellipses) represent sets; we restrict the use of rectangles to given (i.e. fully-abstract) or pre-defined types. Subsets, disjoint sets and set intersections are represented visually: by containment, disjointness or intersection of contours, respectively. Contours partition each diagram into minimal regions or zones; a region is any combination of (one or more) zones.

The dots and asterisk are examples of spiders, which represent elements of a set corresponding to the region that they inhabit. A dot is an existential spider denoting some element of this set, and an asterisk is a universal spider denoting each element of this set. Distinct spiders in a region denote different elements. A shaded zone containing no spiders represents the empty set.

Contours and spiders can be labeled, to identify the sets or elements that they represent. Identical labels are equated when diagrams are conjoined.

A labeled arrow represents a binary relation, where its source and target can be either a spider or a contour. This target corresponds to the image of an element or set represented by its source, under a relation which is identified by the label appearing on that arrow.

2.1.1 Queries

Named operations that may be applied to objects of a class leaving their state unchanged are introduced by the ‘?’ separator. Each query is specified in terms of a pre-condition alone. This is conjoined with the (undashed) invariant for that class, which constrains the types and values of all state-variables.

One query that may well be required at this level is: VM?showInfo(m → i)

This shows the information i for a current member m. Both dots are labeled in the pre-condition, to establish a correspondence with its input- and output-arguments; their types follow from the invariant. Another query is: VM?showMembers(→ M)

M := Info

This shows all member identifiers, with their associated information; its result M is equal to the current value of function Info, which is fully specified by the invariant.

2.1.2 Events

Named operations that may be applied to objects of a class to change their state are introduced by the ‘!’ separator. Each event is normally specified in terms of both a pre-condition and a post-condition (below the double line). These are conjoined with the (undashed and dashed) invariants for that class, which constrain the values of its state-variables before and after any allowable occurrence.

At least one event is obviously required at this level, to add a new member m with associated information i:

VM!NewMember(i → m)

The pre-condition ensures that its input-argument i has type I, and that its output-argument m is an identifier of type M which is not in Memb. In the post-condition, dashed names denote values that are changed – but only minimal changes are shown here; there is no need to say that other elements of Memb and their associated Info values remain the same, because of our convention that “the rest stays unchanged.”