Visualising Action Contracts in Object-Oriented Modelling

Stuart Kent
Division of Computing,
University of Brighton, Lewes Rd., Brighton, UK.
http://www.it.brighton.ac.uk/staff/Stuart.Kent/,
Stuart.Kent@brighton.ac.uk
fax: ++44 1273 642405, tel: ++44 1273 642494

Joseph (Yossi) Gil
IBM T.J. Watson Research Center,
P.O. Box 704,
Yorktown Heights, NY 10598, USA.
yogi@cs.technion.ac.il

Abstract. In recent years a number of OO methods have been enhanced with textual, mathematical languages for specifying invariants and action contracts (pre and post conditions). This paper builds on a recent proposal for “constraint diagrams”, a diagrammatic notation allowing the expression of such assertions. Constraint diagrams essentially provide a pictorial representation of navigation expressions, specifically the sets of objects they define, and, using Venn diagrams and other techniques, constraints on the cardinalities of and relationships between those sets. The original proposal focused on the use of constraint diagrams for depicting invariants. This paper focuses on their use in depicting action contracts.

1 Introduction

Diagrammatic notations for software modelling are currently being standardised under the Unified Modelling Language banner (UML Consortium, 1997). UML derives much of its inheritance by the advances made in Object-Oriented Software Modelling, in particular the OMT (Rumbaugh et al., 1991) and Booch (Booch, 1991) methods. However, it has long been recognised that these notations are not sufficiently expressive without the inclusion of some textual annotation language.

A number of methods, have proposed precise, mathematically-based textual languages for writing these annotations for many of the same reasons that have inspired the study of Formal Methods. Particularly influential have been Syntropy (Cook & Daniels, 1994) and Catalysis (D’Souza & Wills, 1996, 1997), many of whose ideas have been adopted in the Object Constraint Language (OCL; UML Consortium, 1997) which is being incorporated into the UML standard.

Perhaps not surprisingly, this work draws on the rich body of research into Formal Methods, most notably Z (Abrial et al., 1980) and VDM (Jones, 1990). The authors have strived to bring the benefits of precision combined with expressiveness and abstraction from Formal Methods into the domain of object-oriented modelling, but in a form which is accessible to the wider community of software developers. Specifically they strip away all but the essentials for writing precisely those constraints which can not be expressed diagrammatically; they add only a notation for navigating around object configurations (see e.g., Hamie et al., 1998a).

This paper describes work which continues this process, by removing the need for a textual language altogether. Following in the tradition of Harel's seminal work on Visual Formalisms (Harel 1987, 1988), Kent (1997) introduced the notation *constraint diagrams* for visualising invariants in Object-Oriented models. It was suggested there that the notation could be used to visualise action contracts, expressed as pre/post conditions. This paper builds on the suggestion, introducing two new forms of diagram: *post-box* and *contract-box*. The paper draws on parallel work in three dimensional notations for modelling software (Gil & Kent, 1998).

Section 2 introduces UML via an example, which is used in the remainder of the paper to illustrate and reinforce the main arguments. Section 3 gives an example of an invariant which must be expressed as a textual annotation to UML diagrams, but which can be shown visually using the *constraint diagram* notation. In so doing it introduces the main elements of the notation. Section 4 is the core of the paper, showing how *post-* and *contract-boxes*, which incorporate constraint diagrams, may be used to visualise action contracts, including those parts which can not be expressed, without significant textual annotations, with state diagrams. An appendix summarising the notation of constraint diagrams, post- and contract-boxes is provided.

2 Unified Modelling Language

Constraint diagrams are intended to be used in conjunction with object-oriented modelling (OOM) notations. Therefore, in order to understand the notation, it helps to understand aspects of existing OOM notations. This section introduces the three most relevant diagrams from UML, the emerging standard language for OOM. It also serves to introduce the running example.

2.1 Class and Object Diagrams

The main diagram in UML is the class diagram. Figure 1 shows the class diagram of a toy library system. The boxes represent classes (types) and the edges, associations. The meaning of the diagram can be explained in terms of the invariant constraints it places on the set of possible object configurations the model may enter.

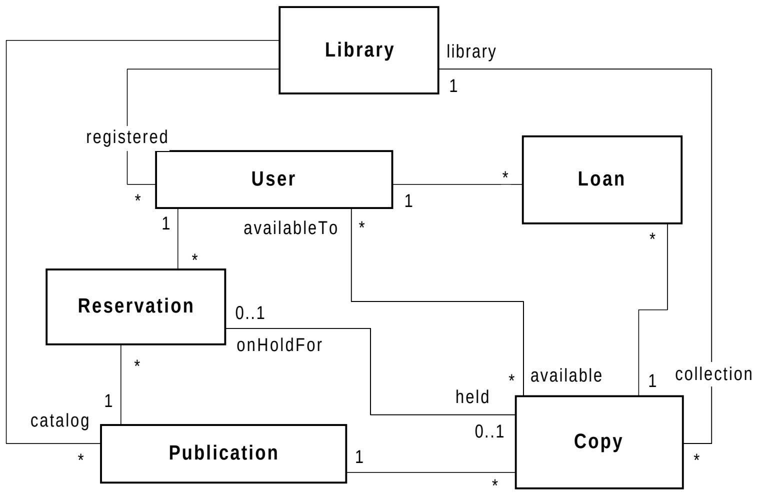


Figure 1: Class diagram for library

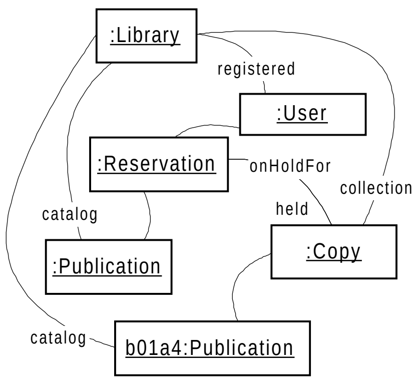


Figure 2: Object diagram

An object configuration of a model is a collection of objects, connected by labelled links. This may be visualised using object diagrams, such as that in Figure 2. Each rectangle represents an object: b01a4:Publication means that the object is of type Publication and has been given the explicit identity b01a4 in this state.[1] The lines represent explicit links between objects, where the links are instances of associations. For example, in this particular object configuration the links labelled catalog from the library object indicate that navigating the association catalog from that object results in a set containing the two publications depicted.

The class diagram says that only objects and links, of the types and with the labels appearing in the diagram, respectively, may be part of the configuration, links must connect objects of the appropriate type, and the cardinalities of associations must be preserved. The cardinalities of an association are indicated by number ranges at either end, where * means many. Thus the association between Publication and Copy indicates that a Publication object may be associated with zero to many Copy objects, but that a Copy object must be associated with exactly one Publication object. You will see that these rules are obeyed in Figure 2, so

1. This particular identity has been chosen to make it clear that these identities have nothing to do with labels identifying attributes and association roles. They are purely there for helping to refer to particular objects when explaining the diagram.

this depicts a valid object configuration for Figure 1 (though not necessarily a valid configuration for the system being modelled – see Section 3, “Constraint Diagrams”).

2.2 State Diagram

State diagrams in UML are based on Harel statecharts (Harel, 1987). They are used to model (in part) dynamic behaviour, specifically how the state of the system changes as it responds to events.

An example state diagram is given in Figure 3. It shows the states of an arbitrary Copy object, and how that object responds to various events. In this case the events are actions on the library system owning the copy.

The diagram indicates that the copy can be in one of two states, Available or Unavailable. Those states are further partitioned into substates. Thus when a copy is available it may either be on hold or on a shelf.

.Transitions run between states. They mean that when the labelled event happens, the object changes state from the source of the transition to the target. Thus if the library associated with the copy performs a return action and the argument to that action is this object, then the copy will move from the Out to the Returned state. When the library performs the clearReturns action it takes all the copies from the return bin and, for each one, either puts it back on the shelf or puts them on hold for a reservation. Actions with under-determined behaviour, such as for ClearReturns, are tolerated in a specification model.

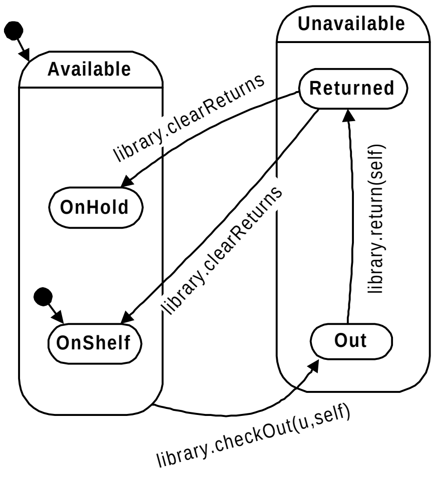


Figure 3: State diagram for Copy

3 Constraint Diagrams

Kent (1997) introduces a new notation, called *constraint diagrams*, which derives its name from an ability to express constraints on a model that can not be expressed using existing diagrammatic notations for object-oriented and structured systems modelling. Constraint diagrams are an application of Harel’s hi-graphs (Harel, 1988) in the domain of specification of object-oriented systems, with several important extensions: the ability to show set membership, and the ability to quantify both universally and existentially over set members, in addition to set containment. These are achieved in part by introducing notation for showing singletons (and sets of other cardinalities).

The notation is still under development: a semantics is being worked out, which is helping to resolve some ambiguities, and every opportunity is being taken to expose the notation to practioners to obtain some feedback on its applicability and utility. We are reasonably convinced that there are no serious semantic flaws, and feedback from practioners tends to be positive (a) after they have had time to get used to it and/or (b) if they are already familiar with and like visual modelling notations.

The notation is summarised here, using the example in Figure 4 for illustration. A more complete definition of the notation as it currently stands is given in the appendix. Figure 4

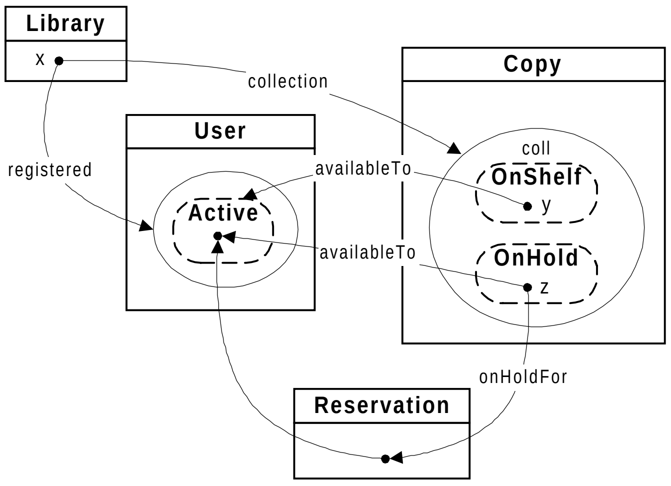


Figure 4: Constraint diagram showing invariant

shows a constraint diagram showing an invariant which can not be expressed with UML diagrams (demonstrated in Kent, 1997), even enhanced with ideas for combining state and class diagrams, such as in D'Souza (1992). The only other way to express the invariant precisely is to annotate a class diagram with an expression of a textual, mathematical language, such as OCL (UML Consortium, 1997).

Informally, the invariant may be stated as follows:

> For any library and any copy in that library's collection, if the copy is on the shelf then it is available for borrowing to all active users registered with the library. If, however, the copy is on hold for a particular reservation that has been made, then it is only available to the registered user that made the reservation.

We now proceed to explain how this invariant can be read from the diagram, and in so doing introduce the main parts of the notation.

Before we begin, it should be noted that the notation is highly dependent on sets. Thus a type is treated as the set of objects of that type and is indicated by the same symbol as a type on a type diagram. A state is treated as the set of objects in that state, and is indicated by the symbol used to represent states on a state diagram. Indeed, enclosing the state diagram of Figure 3 in a type box labelled Copy, and removing the transition arrows, would result in a constraint diagram. Other sets are introduced below. The other main piece of notation are links, which correspond to associations on type diagrams.

The starting point for reading Figure 4 is x. This is an arbitrary singleton set (i.e., object) of the type Library. The fact that it is a singleton is indicated by a small filled circle. Special symbols exist for sets of other cardinalities; they are detailed in the Appendix.

y is an arbitrary object of the set that is the intersection of the set of objects in the state OnShelf and the set coll, which, by the link from x, represents the collection of copies for x. (coll could instead have been referred to using the navigation expression x.collection.) This is indicated by placing y inside the projection of OnShelf, shown by using a dashed line for the boundary. A projection is defined to be the intersection of the set being projected (in this case the state OnShelf) with the smallest set that contains it, in this case x.collection.

Similarly, z is an arbitrary Copy object which is OnHold and in the library's collection. Thus so far we have read the following part of the invariant:

> For any library and any copy in that library's collection, if the copy is on the shelf then …. If, however, the copy is on hold for a particular reservation that had been made, then….

The rest of the invariant comes from following the links from y and z. Following from y, we see that the users to which y is available is the projection of Active into the set x.registered, i.e., all active users registered with the library. Thus we derive the constraint that y is available to all active users registered with the library. Following the links from z, we see that the set at the end of the availableTo link (z.availableTo) is a single object, and that this is the same as if we had followed the onHoldFor link to the reservation it is on hold for, and then the link to the user associated with the reservation. In our model this is the user who made the reservation. Hence z is only available to the (active) user who made the reservation it is on hold for. It is a small step to rewrite the invariant in its final form from the partial invariant above and the constraints on y and z.

The equivalent of the invariant in OCL is:

```
Library
    collection->forAll( c |
        (c.oclIsKindOf(OnHold) implies c.availableTo = c.onHoldFor.user)
    and
        (c.oclIsKindOf(OnShelf) implies
            c.availableTo = registered->select(oclIsKindOf(Active))) )
```

where Library indicates that this is an invariant which applies to Library objects. x.oclIsKindOf(T) is true when the type of x is conformant with T. By treating states as dynamic types in UML, this is how we can state that an object is in a particular state. he full notation for constraint diagrams, summarised in the appendix, also includes symbols for showing inheritance relationships between types, nesting of states and existentially quantified elements.

4 Action Contracts

4.1 State Diagrams aren't enough

State diagrams are not sufficiently expressive to capture the complete behaviour specifications of actions. As with invariants, this has led to a range of proposals for more or less precise textual languages to complement the visual notations. For example, Syntropy (Cook & Daniels, 1994) proposes the use of a Z-like language for annotating transitions on state diagrams; Catalysis (D'Souza & Wills, 1996, 1997) proposes a language for writing separate pre and post conditions on actions, and this is the position also adopted within UML/OCL.

To understand why textual annotations are needed, consider again the state diagram of Figure 3. Many things are missing from the specified behaviour. For example, considering the checkOut action, the state diagram only states that the copy being checked out moves into the Out state. It doesn't state that a new loan object is created, recording the fact that the copy is checked out to user u. Even providing another state diagram for Loan won't resolve the problem: the most that can be stated on such a diagram is that a loan object is created by some library's checkOut action, and when created is initialised to some state.

The additional behaviour may be written precisely as follows:

```
Library::checkOut(u:User, c:Copy)
    pre
        c.availableTo->includes(u) and collection->includes(c)
    post
    c is no longer available for lending.
        c.availableTo->isEmpty
    The loan of c to u is recorded and marked as ongoing.
        Loan.allInstances->exists( l | Loan.allInstances@pre->includes(l) and
            l.oclIsKindOf(Ongoing) and l.user=u and l.copy=c )
```

l.oclIsKindOf(Ongoing) could be omitted, if a state diagram was also provided for Loan. Library:: indicates that this is a fragment of specification for the checkOut action on Library objects.

A state diagram can be regarded as just a means of visualising some aspects of what can be written textually. For the checkOut transition in Figure 3, the textual equivalent may be derived as follows. First write the specification from the point of view of a Copy object:

```
Copy::library.checkOut(u:User, self)
    pre
        oclIsKindOf(Available)
    post
        oclIsKindOf(Out)
```

Then convert this to a specification on a Library object, by replacing self with the formal argument of checkOut, noting that self is generally omitted from the start of navigation expressions (i.e. oclIsKindOf(...) is equivalent to self.oclIsKindOf(...)):

```
Library::checkOut(u:User, c:Copy)
    pre
        c.oclIsKindOf(Available)
    post
        c.oclIsKindOf(Out)
```

The full specification can then be obtained by composing this with the very first fragment: the pre- and post-conditions, respectively, are conjoined.[1]

In Figure 3 there is only one transition labelled checkOut, so this approach is equivalent to adding the annotations directly to the state diagram, as in Syntropy, where pre conditions get re-termed *guards*. If there are many transitions with the same label, then some indication must be given as to which fragment of specification should be composed with the specification derived from which transition, where a fragment could be combined with many transitions. Syntropy allows a specification fragment to be combined with a single transition (the transition is annotated) or to all transitions (the specification is written separately). Catalysis proposes that compositions should be written explicitly.

To summarise, state diagrams can only be used to visualise part of an action specification, the rest must be written textually. It is largely immaterial whether these textual fragments appear as annotations on the state diagram or are written separately. It is also possible to write the part of the specification contributed by a state diagram textually, and this may be composed with any other fragments of specification to produce the complete contract for an action.

In the remainder of this section we show how constraint diagrams may be used to visualise any kind of action contract, whether visualised by a state diagram or not and whether a fragment or a complete specification.

1. Another form of action composition is what has been dubbed in d'Souza & Wills *angelic* composition. Here pre-conditions are disjoined and the post-condition takes the form (pre1 implies post1) and (pre2 implies post2). Angelic composition is used e.g. when deriving the specification of an action by composing fragments derived from different transitions labelled by that action in the state diagram of any one particular type.

4.2 Pre-conditions

The pre-condition of an action contract is visualised by a single constraint diagram, which is little different from visualising invariants. There are two additions to the notation:

- An icon for action arguments. The particular icon chosen is, in a sense, irrelevant. We have chosen a lozenge, though realise that a different icon (e.g. a 3D one) may be more appropriate, especially if we were able to draw constraint diagram pairs (see below) in true 3D.
- A labelled lightning bolt arrow targeted on a particular object, indicating the action to whose pre-condition the diagram contributes.

The notation is illustrated by the pre-condition constraint diagram for checkOut in Figure 5, which visualises the textual pre-condition given in the previous section.

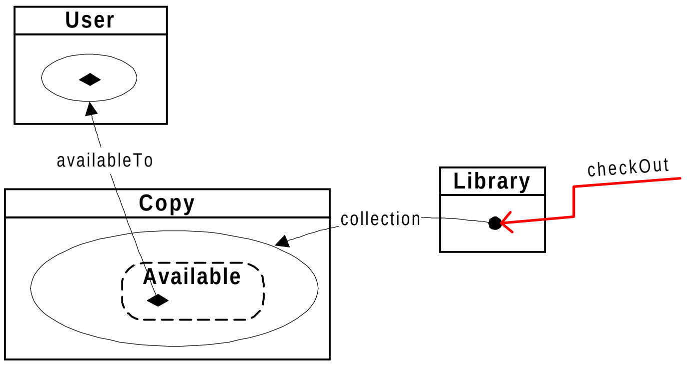


Figure 5: Pre-condition of checkOut

4.3 Post-conditions: Post-Box

A post-condition is expressed using two constraint diagrams, one to refer to the before or pre state and one to refer to the after or post state. Two additional pieces of notation are required.

- Lifelines which are used to link sets between the two diagrams. The meaning of a lifeline is that the two sets so connected contain exactly the same objects. That is the line shows the life of a set objects as it changes its relationships with other sets.
- A symbol (here a ☆) to show the creation objects.

As discussed in Gil and Kent (1998), a 2D view of the constraint diagram pair, e.g. with one on top of the other, can become very cluttered with the addition of lifelines. This visual clutter is reduced if the diagrams are rendered in 3D, or, as is done in this paper, a stereoscopic projection of the true 3D image. Essentially the third dimension indicates more clearly how to conceptually break up the diagram into separate units of comprehension, hence making it easier to comprehend as a whole.

An example of a post-box is given in Figure 6. Focusing on the bottom constraint diagram, we see that the result of the action is to create a single new loan object, indicated by ★,

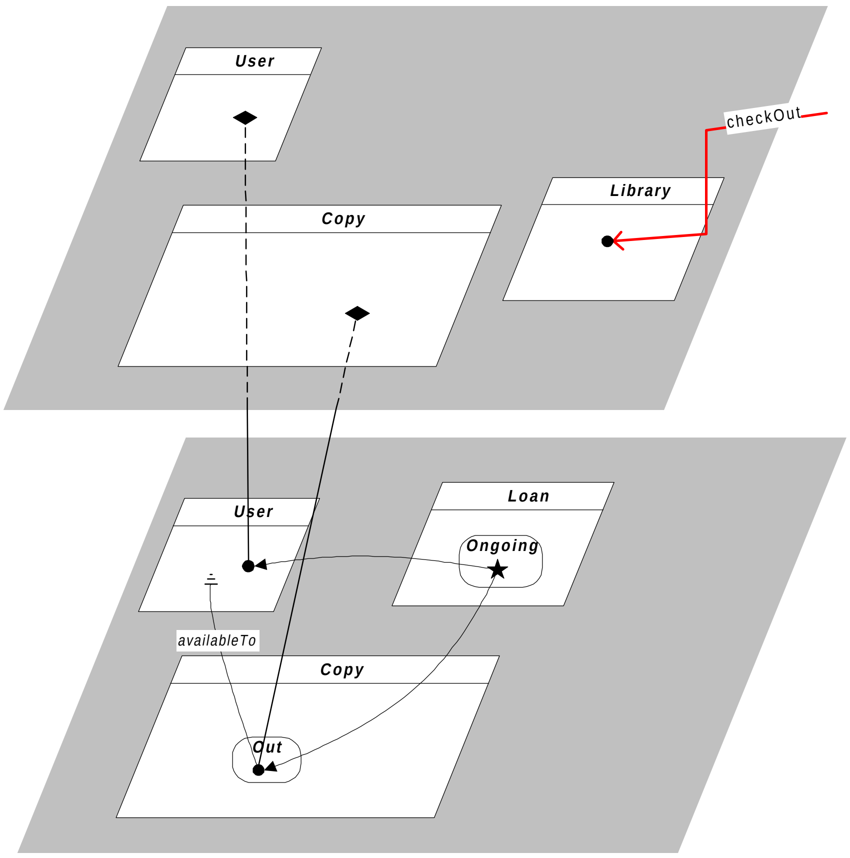


Figure 6: Post-box for checkOut

which is in the Ongoing state. This is associated with the user and the copy provided as arguments to the action; we know this by the lifelines connecting the objects involved in these associations to the objects supplied as arguments in the top constraint diagram. That same copy object goes into the Out state and thus is not available to any users, as indicated by the availableTo link which has the null pointer symbol $\doteq$ for an arrowhead.[1]

4.4 Combining pre & post: Contract-Box

It is possible to overlay the pre-condition constraint diagram on top of a post-box, resulting in a *contract-box*. In a tool, colours could be used to distinguish between the contributions made by the different diagrams. The contract-box for checkOut is shown in Figure 7.

1. This would be redundant if there was already an invariant stating that a copy that is out is unavailable. This is based on the assumption that the specifications assume an implicit frame rule that, provided the action is performed in isolation, nothing changes unless changed directly by the post-condition of the action, or indirectly by that post-condition in order to preserve invariants. Thus we adopt a 'minimal specification' approach, where we specify exactly what changes and no more.

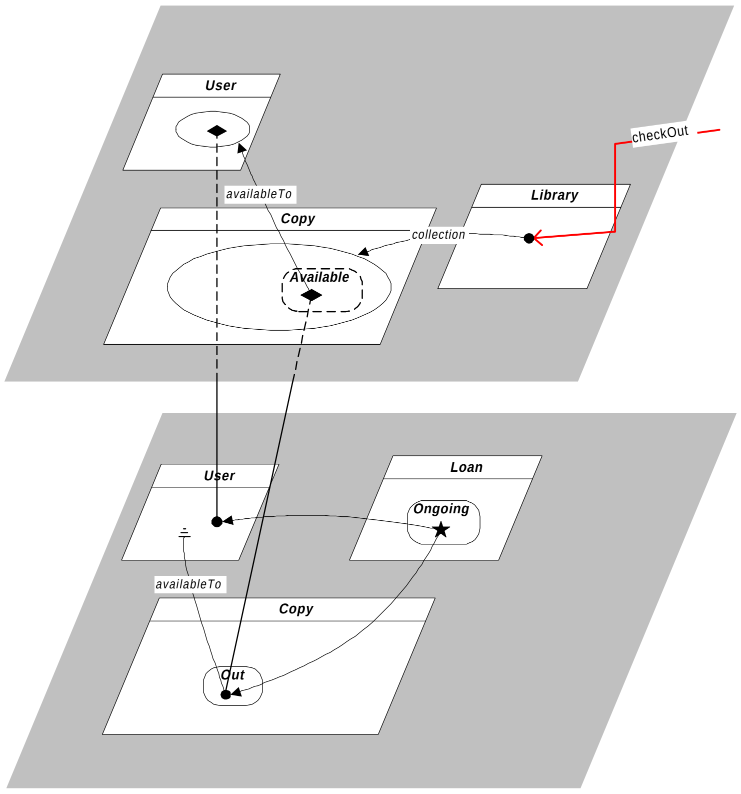


Figure 7: Contract-box for checkOut

4.5 State Diagrams and Contract Boxes

Transitions on state diagrams are projections from contract boxes. They would be constructed as follows. Focus on the objects that change state, in this case a single copy object. For each of these objects, draw a transition on the Copy state diagram with a single transition labelled with the name of the action being performed, preceded by a navigation expression which locates the object on which the action is invoked with respect to the object under consideration, and with arguments, which are also similarly located. In this case the label is library.checkOut(u,self): the action being performed is checkOut on the library object obtained by navigating the library link (same as backwards down the collection link) from the copy object under consideration; the arguments, from the point of view of the copy, are some arbitrary user and the copy itself, which is labelled self. The source of the transition is the state in which the copy object appears on the top diagram, and the target is the state in which it appears in the bottom diagram.

Rather than visualise the complete specification of checkOut using a single constraint diagram, we could have used a constraint diagram to visualise only that part which is not already visualised through the state diagram. In which case all sets representing states would be removed from Figure 7. Composition of the two specifications could be done as outlined earlier by deriving the textual specifications from the constraint diagrams and state diagrams, and then composing those. However, then it might not be so clear how to generate the visualisation of the complete specification (i.e., Figure 7).

An alternative approach would be to bypass the textual specification altogether, by deriving contract boxes from the state diagram transitions, then composing those with the contract box representing the additional component of specification. This requires a calculus for composing constraint diagrams and contract boxes, which remains to be worked out.

The use of contract-boxes for additive specification, i.e. only to express those aspects not already expressed by the state diagram, is not always possible as the example in the sequel illustrates.

4.6 Referring to the pre state

Another common feature of precise action contracts is the use of an operator in the post-condition to enable reference to elements of the pre state. In post-boxes this reference is achieved by the lifelines. For example, imagine a clearReturns action on the library, which is used to allocate copies that have been returned either to go back on the shelf or to be put on hold for a reservation that has been made for the appropriate publication. There are many ways of performing such an allocation. For example, suppose three copies of “War and Peace” have been returned and there are two reservations for this book. How does one allocate copies to fulfil the reservations? In order to spread wear and tear across different copies, the allocation might be done based on the number of times the copies have been previously loaned out; or it might be done on the basis of which comes first out of the return bin. What if there are more reservations than there are copies returned? Are copies allocated according to oldest reservation, or is there some other priority system in place? Whilst these decisions have to be made at some point, it is probably not a good idea to make them at the top level specification. Instead, we just ensure that, for any publication, the correct number of copies are put on hold and the correct number are put on the shelf.

The detailed specification for this post-condition is given by Figure 8. In the top constraint diagram, the sets of copies and reservations of interest for the publication under consideration, namely those copies returned and those reservations waiting to be allocated a copy (i.e., pending), are identified. Lifelines are used to take these sets into the constraint diagram representing the after state, where the copies are separated between those put on the shelf and those put on hold; similarly for the reservations.

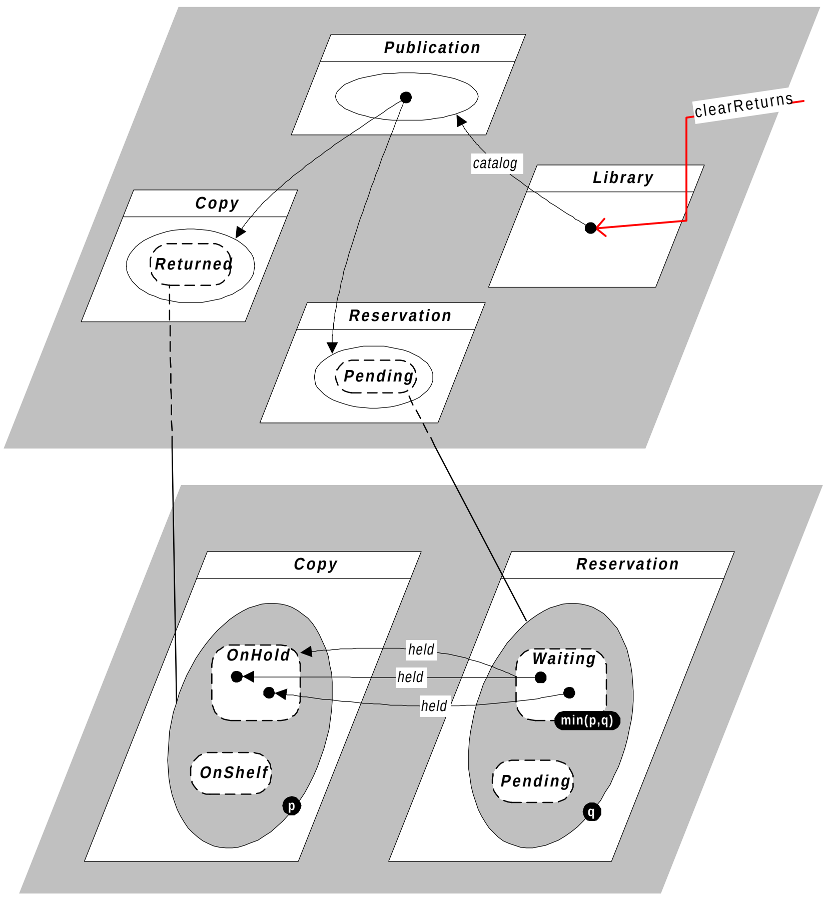


Figure 8: Post-box for clearReturns

Textually, the lifelines correspond to the use of @pre in OCL, as in the following specification fragment derived from this post-box.

```
catalog@pre->forAll( p |
    ( (p.reservations->select(oclIsKindOf(Pending)))@pre
            ->select(oclIsKindOf(Waiting)) )->size
        = (p.copies->select(oclIsKindOf(Returned)))@pre->size.
            min((p.reservations->select(oclIsKindOf(Pending)))@pre->size)
    and ( (p.reservations->select(oclIsKindOf(Pending)))@pre
        ->select(oclIsKindOf(Waiting)) )
            ->forAll( r1 | ( (p.reservations->select(oclIsKindOf(Pending)))@pre
                ->select(oclIsKindOf(Waiting)) )
                    ->forAll( r2 | not r1 = r2 implies not r1.held = r2.held ) )
    and (p.reservations->select(oclIsKindOf(Pending)))@pre.held =
        (p.copies->select(oclIsKindOf(Returned)))@pre
    and (p.reservations->select(oclIsKindOf(Pending)))@pre –
        (p.reservations->select(oclIsKindOf(Pending)))@pre
            ->select(oclIsKindOf(Waiting))
        = (p.reservations->select(oclIsKindOf(Pending)))@pre
            ->select(oclIsKindOf(Pending))
    and (p.copies->select(oclIsKindOf(Returned)))@pre –
        (p.copies->select(oclIsKindOf(Returned)))@pre
```

```
                ->select(oclIsKindOf(OnHold))
            = (p.copies->select(oclIsKindOf(Returned)))@pre
                ->select(oclIsKindOf(OnShelf))
        )
```

4.7 Under-determined Specification

The detail in the bottom constraint diagram in Figure 8, shows that the number of Waiting reservations, with copies waiting to be collected, is the smallest of the number of reservations that were originally pending (q) and the number of copies that were returned (p) for the publication under consideration. The links between the Copy sets and Reservation sets, ensure that there is a 1-1 and onto relationship between the set of waiting reservations and the set of copies on hold, to ensure that a copy is not put on hold for two reservations simultaneously and that the only copies put on hold are those that are put on hold for a waiting reservation. This also ensures that the cardinalities of the two sets are equal.

This specification is under-determined in the sense that it does not say exactly which copies are put on hold or which reservations move from pending to waiting. On a state diagram such under-determinism is visualised by showing two transitions from the same state and with the same label targeted on different states, as is the case for clearReturns in Figure 3. Furthermore, the use of post/contract-boxes as a means of additive specification is not possible in this case. This is typified by, for example, any attempt to add guards to the transitions in the state diagram. For example, on the transition targeted on the OnHold state, a guard would be required stating the conditions under which the copy is put on hold. These conditions can not be specified, because the specification is under-determined – for any particular copy; it is not possible in general to predict whether it will be put on hold or not. The specification can only be written in terms of the sets of copies involved, not individual copies. This highlights the power of constraint diagrams, which work with sets of objects, over state diagrams, which can only express properties of individual objects.

5 Conclusions

The paper has given a brief tour of the constraint diagram notation, then incorporated this into post- and contract-boxes which visualise action contracts. The paper showed (by example) that the notation was more expressive than state diagrams. It is hoped the reader is convinced that intuitiveness has not been sacrificed for expressiveness.

The notation is only at the first stage of its development, to be regarded as a true Visual Formalism, in Harel’s sense. The next stage requires semantics work to ensure the notation’s integrity and to underpin the provision of CASE tool support.

For the semantics we are building on work (Hamie et al. 1998a, 1998b) which uses Larch (Guttag & Horning, 1993) to give a semantics to OO modelling notations, including, recently, a fragment of UML which includes OCL. Work in diagrammatic reasoning (Allwein and Barwise, 1996), in particular with regard to Venn diagrams (Hammer, 1996), is also relevant. So far this work has uncovered some ambiguities in the notation, but nothing that it has not been possible to fix.

We envisage the semantics work contributing to the development of tools for integrity checking of diagrams, diagram composition, model simulation (visualised through the diagrams), generation of specifications (as constraint diagrams and post/contract-boxes) from object diagrams and vice-versa, etc.

Work is progressing on integrating the notation better with UML. Specifically, we see how constraint diagrams could be integrated with sequence and collaboration diagrams, which would allow these to be used in a more generic way. An offshoot of this effort is a series of 3D notations for visualising models (Gil & Kent, 1998), which, for example, includes a 3D sequence diagram integrating collaboration, sequence and constraint diagrams. The hope is that this will lead to a fully integrated notation that, given appropriate CASE tool support, would allow a model to be visualised in one 3D diagram that could be explored by zooming and projection capability.

No attempt has yet been made to formally evaluate the applicability or utility of the notation. We feel that this would be premature without some of the aforementioned semantics work and CASE tools in place. However, informal feedback from practioners has generally been positive, with the caveats that (a) sufficient time is allowed for them to get used to the notation, and (b) they already accept the utility of visual modelling notations. Specifically, in our own experiments we have noticed that the notations tend to lead to more complete specifications faster.

Acknowledgements

Thanks for comments and general support are due to colleagues on the BIRO project at Brighton, in particular Franco Civello, Ali Hamie, John Howse and Richard Mitchell. Stuart Kent is partially funded by the UK EPSRC under grant number GR/K67304.

References

Abrial, J-R., Schuman, S. and Meyer, B. A Specification Language. On the Construction of Programs, McNaughten R. and McKeag R. (eds.), Cambridge University Press, 1980.

Allwein, G. and Barwise, J. (eds) Logical Reasoning with Diagrams. Oxford University Press, 1996.

Booch, G. Object-0riented Design with Applications. Benjamin Cummings, Redwood City, California, 1991.

Cook, S. and Daniels, J. Designing Object Systems: Object-Oriented Modelling with Syntropy. Prentice Hall, 1994.

D'Souza, D. & Wills, A. Extending Fusion: practical rigor and refinement. R. Malan et al., OO Development at Work, Prentice Hall, 1996.

D'Souza, D. and Wills, A. Objects, Components and Frameworks with UML: The Catalysis Approach. Addison-Wesley, to appear 1998. Draft and other related material available at http://www.trireme.com/catalysis.

D'Souza, D. Education and Training: Teacher! Teacher!. Journal of Object-Oriented Programming 5(2):12-17, 1992.

Fowler, M. with Scott, K. UML Distilled. Addison-Wesley, 1997.

Gil, Y. and Kent, S. Three Dimensional Software Modelling. Proceedings of ICSE98, IEEE Press, to appear 1998.

Guttag, J. and Horning, J. Larch: Languages and Tools for Formal Specifications. Springer-Verlag, 1993.

Hamie, A., Howse, J. and Kent, S. Navigation Expressions in Object-Oriented Modelling Notations. Proceedings of FASE98 in ETAPS98, Springer Verlag, to appear, 1998.

Hamie, A., Howse, J. and Kent, S. Compositional Semantics of Object-Oriented Modelling Notations. Evans, A. and Lano, K., Making Object-Oriented Methods more Rigorous, LNCS Series, Springer Verlag, to appear 1998.

Hammer, E. and Danner, N. Towards a Model Theory of Venn Diagrams. In Allwein G. and Barwise J., Logical Reasoning with Diagrams, Oxford University Press, 1996.

Harel, D. On Visual Formalisms. Communications of the ACM, 31(5)514-530, May 1988.

Harel, D. Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming, 8:231-274, 1987.

Jones, C. Systematic Software Development using VDM (2nd edition). Prentice Hall, 1990.

Kent, S. Constraint Diagrams: Visualising Invariants in Object-Oriented Models. Procs. of OOPSLA97, ACM Press, 1997.

Kent, S., Hamie, A., Howse, J., Civello, F. and Mitchell, R. Semantics Through Pictures. ECOOP'97 workshop reader, LNCS series, Springer Verlag, to appear 1998.

Rumbaugh, J., Blaha, M., Premerali, W., Eddy, F. and Lorensen, W. Object-Oriented Modelling and Design. Prentice Hall, 1991.

UML Consortium. Object Constraint Language Specification Version 1.1. Available from http://www.rational.com, 1997.

UML Consortium. The Unified Modeling Language Version 1.1. Available from http://www.rational.com, 1997.

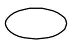
Appendix – Summary of Notation

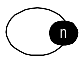
Constraint Diagrams

Normal Sets

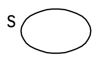
● A set with one element.

○ A set with 0..1 elements.


A set with 0, 1 or more elements.



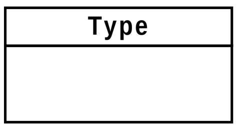
A set with n elements; n may be any numerical expression.

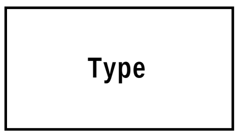


Optionally sets can be explicitly labeled. This can be useful for referring to them in accompanying explanations, or when mapping a constraint diagram to a math expression.

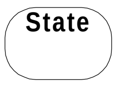
Standard Venn diagram notation may be used to show relationships between sets.

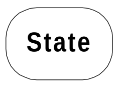
Types and States



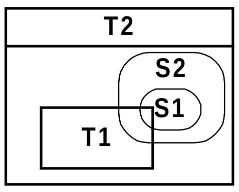
or


The set of objects of the named type Type.



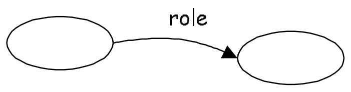
or


The set of objects in state State.

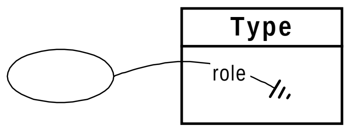


T1 (S1) is a subtype(state) of T2 (S2). T1 intersects with the two states to indicate that T1 objects may enter states S1 and S2.

Navigation



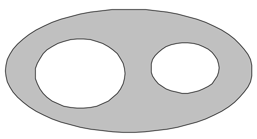
The set at the target of the arrow is the union of the sets reached by navigating the association labeled role from each element in the set at the source.

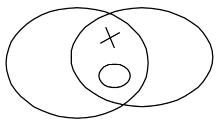


The value of the association to Type and labeled role, when navigated from the source set, is the empty set or null.

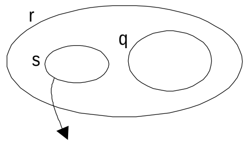
Areas

With quantified sets it can be ambiguous in which area the set is contained. We adopt the following conventions.

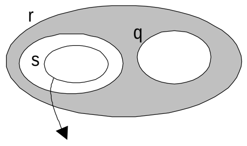

Grey fill indicates that there are no elements in that area of the set. In this case this means that the two subsets partition the containing set.



In those cases where grey fill is difficult to achieve (e.g., by hand or with some drawing tools), a simple cross in the area may be used as an alternative. In this case, the cross means that the set contained in the intersection *is* the intersection – particularly useful for sourcing/targeting arrows from/to an intersection.

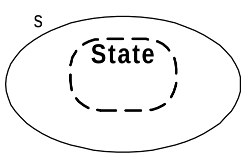


s is contained in r, not necessarily r-q.

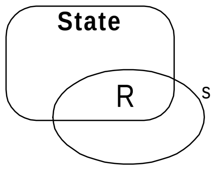


s is contained in r-q.

Projection



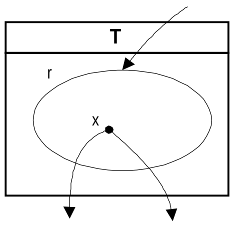
is equivalent to



Sets other than states can also be projected. The only requirement is a label to identify the set.

Universally quantified sets

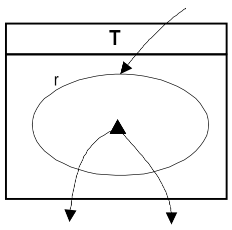
A singleton set with no arrows targeted on it is assumed to be an arbitrary object of the set in which it is contained. When converted to a math expression this translates to universal quantification: “for any object in the containing set, ...”. As usual, labels for sets (other than types or states) are optional.



r->forAll(x | ...)

Existentially quantified sets

The icon ▲ is used to distinguish existential from universal quantification objects. The same context restrictions as universally quantified sets apply.

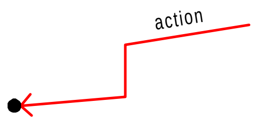


r->exists(x | ...)

Lifelines

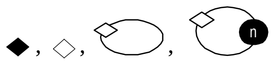
Lifelines connect sets in the diagrams at the top and bottom of the boxes. These indicate that the sets connected have exactly the same contents.

Action invocation



action is invoked on the targeted object.

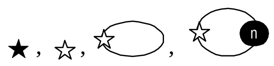
Action arguments



Sets of different cardinality, which are arguments to actions.

New objects

Each of the following symbols represents a set of objects that did not exist in the previous frame. The symbol differs depending on the cardinality of the set.



Sets of different cardinality containing new objects.