Validation of Object Oriented Models using Animation

Ian Oliver, Stuart Kent
University of Kent at Canterbury
England, UK
ian.oliver@bcs.org.uk
http://www.cs.ukc.ac.uk/people/rpg/ijo1
Tel/Fax: +44 (0)1227 764000 / 762811

Abstract

Experience has shown that prototyping is a valuable technique in the validation of designs. However, the prototype(s) can be too far semantically removed from the design. Animation is a technique where a design itself can be 'executed' without the need to translate to a high-level language to build a prototype. While animation has been implemented with formal specification languages such as VDM, Z and B and used with some success, we feel that its application to a more graphical specification language/notation would introduce animation to a wider range of software designers. This paper discusses the basis of a technique for the animation of rigorously specified object-oriented models written using the Unified Modelling Language and the Object Constraint Language.

1. Introduction

When validating designs, techniques such as prototyping have been used to ensure that the model being constructed is "what the customer wants." A prototype for demonstration is constructed by translating the model into a high-level programming language - the success of this depends on the accuracy of that translation.

Studies such as those described in [18] have shown that prototyping reduces the number of problems encountered during the course of the design of a software system. However prototyping has the disadvantage that a prototypical piece of software may bear very little or no resemblance to the model on 'paper' and so changes to the prototype, suggested by the 'customer' may be difficult to transfer in reverse back to the model.

Animation [4] is a technique where the model, specified using a formal specification language, can be made executable in some sense without translating it to a high-level language. It has been shown to be a valuable technique in assisting the validation process [2, 6, 20] and has been implemented in a number of tools [5, 7, 8, 10] supporting traditional formal methods [1, 9, 12, 19].

In this paper we port the ideas of animation to models written using a subset of the UML/OCL. A carefully chosen subset of the UML and the OCL [15, 16] provides a suitable level of formality while being accessible enough for widespread use. Introduced in §2 are the UML and OCL notations, §3 describes an example animation and discusses the issues and problems presented by such an animation. Finally in §4 we discuss future work being undertaken.

2. Notation

In the following library system example, we use a sub-set of the UML/OCL to represent: class diagrams, snapshots, filmstrips, invariants and actions. A fuller explanation of the notation and its semantics can be found in [17].

As part of this work we have written an abstract syntax of the subset of UML/OCL described here and a set of axioms that ensure well-formedness and consistency of the models - see [14] for details.

2.1. Class Diagrams, Invariants and Snapshots

Figure 1 presents the class diagram for our example library system. The class diagram fixes the basic structure of a model. It states what classes of object may appear in an instance of the

model, which classes of object may be linked to one another, and how many links to objects there may be from any one object.

An instance of the model is represented by a snapshot, which is a collection of objects with links between them. Instances must conform to the class diagram. For example, figure 2 has a Library object, two user objects, two loans (one current, one returned), a publication and two copies of that publication. Thus it only has objects of classes shown on the class diagram. In contrast, figure 3 is invalid because it has an object of class Giraffe, which does not appear in the class diagram.

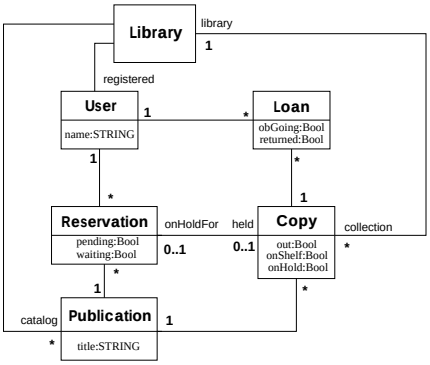

Figure 1. Library System Class Diagram

Invariants - written using OCL - impose further constraints that can not be expressed on the class diagram - four invariants are given below:

context Copy
inv: -- a copy may be out, on the shelf or on hold
out **xor** $onShelf$ **xor** $onHold$

context Loan
inv: -- a loan can be onGoing or returned, not both
$onGoing$ **xor** $returned$

context Reservation
inv: -- a reservation can be pending or waiting, not both
$pending$ **xor** $waiting$

context Copy
inv: -- one reservation may be 'held' at any time per copy
self.$onHoldFor \rightarrow$ **notEmpty**
implies self.$onHoldFor \rightarrow$ **size** $= 1$

An invariant is a constraint in the context of a particular class. The invariant reads; "for any object in the context *Class*, the following constraint holds..." Close inspection of figure 2 will reveal that it does satisfy the invariants given above. However, figure 3 does not, for example: *l1* breaks the loan invariant by having both attributes *returned* and *onGoing* set to true.

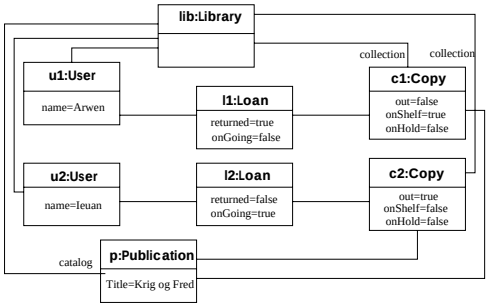

Figure 2. Example Snapshot

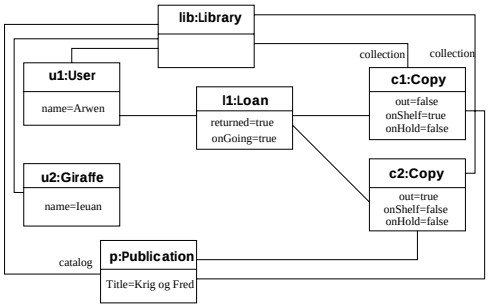

Figure 3. Example Invalid Snapshot

2.2. Filmstrips and Actions

While a snapshot describes a specific instance of a model, a filmstrip is the instance of an action. Descriptions of the action for the filmstrip in figure 4 are given below.

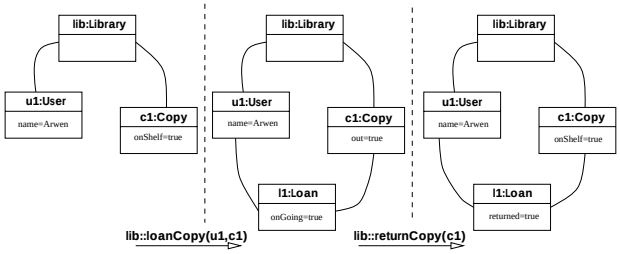

Figure 4. Example Filmstrip

context Library::loanCopy(u:User,c:Copy)
pre: $c.onShelf =$ **true**
post: $c.out =$ **true** **and** $c.loan \rightarrow$ **existsNew** $(l |$
$l.onGoing = true$ **and** $l.user = u)$

context Library::returnCopy(c:Copy)

pre: $c.out = \textbf{true}$

post: $c.loan \rightarrow \textbf{select}\ (onGoing = \textbf{true}\ ).returned = \textbf{true}$
and
$c.publication.reservation \rightarrow \textbf{select}\ (pending = \textbf{true}\ )\textbf{@pre}$
$\rightarrow \textbf{notEmpty}\ \ \textbf{implies}\ (\ c.publication.reservation \rightarrow$
$\textbf{select}\ (pending = \textbf{true}\ )\textbf{@pre} \rightarrow$
$\textbf{exists}\ (r|r.waiting = \textbf{true}\ \ \textbf{and}\ r.held = c)$
and
$c.onHold = \textbf{true}\ )$
and
$c.publication.reservation \rightarrow$
$\textbf{select}\ (pending = \textbf{true}\ )\textbf{@pre} \rightarrow$
$\textbf{isEmpty}\ \ \textbf{implies}\ c.onShelf = \textbf{true}$

An action description indicates what kind of object can perform the action and with what arguments. For example, the first description indicates that Library objects may perform the loanCopy action, with a user and a copy as arguments. It also provides a constraint on the state of the system that must hold when the action is performed (pre-condition) and what the result of performing that action will be (post-condition) [11].

On the filmstrip, an instance of this action appears. It is an instance, because represents the action being performed on a particular object (lib) with particular arguments ($u1$, $c1$) from a particular starting state (the first snapshot in the sequence). That snapshot must satisfy the pre-condition for the action, i.e. $c1$ must be on the shelf, and the second snapshot in the sequence must satisfy the post-condition: a new loan object linked to $u1$ and $c1$ must have been created.

The example shown here is deterministic. Provided one follows the principle that nothing else changes (the so-called frame rule [3]), the only snapshot that can be reached by performing $lib.loanCopy(u1, c1)$ is the second snapshot in the sequence. However, this is not always the case: the post-condition of $returnCopy$ actions is such that a range of different snapshots could be generated from any instance of the action.

The goal of animation is to generate the post-snapshot(s) for a particular instance of an action: an invocation of that action on a particular object, in a particular snapshot of the system state, and with particular substitutions for the arguments. In developing an automated animation technique the two biggest problems are: dealing with non-determinism and ensuring that a sensible frame rule is applied. In the next section, we describe, by example, the basis for a technique that could provide some automatic assistance for the animation of a model.

3. A Technique for Animating UML/OCL Models

Figure 5 shows the expected results of performing the $returnCopy$ action in a snapshot where there are two reservations for the publication of the copy which is being returned. After the application of the action, either of the two snapshots indicated would be reasonable results: the only difference between them is that the copy has been put on hold for a different reservation. We would argue that no other snapshots would be regarded as reasonable, by application of a sensible frame rule.

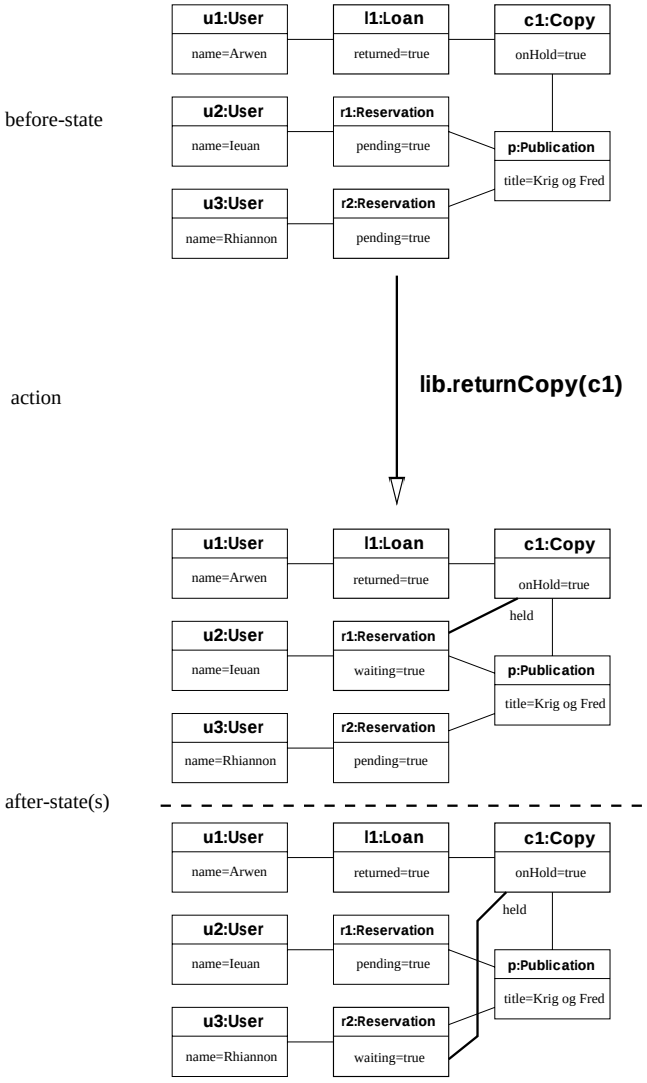

Figure 5. Animation of $lib.returnCopy(c1)$

We now examine more closely the steps that might be in-

volved in an algorithm that would ensure this result. We envisage three basic steps: calculating the post-condition execution paths, evaluating the effects of each OCL term in the various execution paths and checking the model against invariants. Each is examined in detail below.

3.1. Processing the OCL Post-condition

Because the post-condition only states what is true after the application of the action it contains no information about any execution order. Animation however relies on some form of ordering of the statements in an expression. To ensure that we animate all possible combinations we must expand the post-condition to express all possible execution paths.

Two clauses related by an **and** connective, e.g: A **and** B may be animated as its permutations: A *then* B and B *then* A. Two clauses related by an **or** connective, e.g: A **or** B may be animated as its combinations: A *then* B, B *then* A, *just* A or *just* B. The **implies** connective acts as a guard to a statement, e.g: A **implies** B is animated as if A *then* B. Using these rules the post-condition of $returnCopy()$ (the anatomy of which is shown in figure 6) may processed thus:

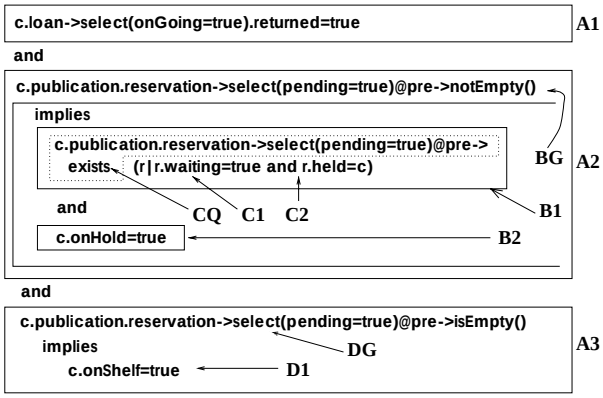

Figure 6. Anatomy of the Post-Condition of $returnCopy()$

$$
\begin{aligned}
&permute([A_1, A_2, A_3]) \\
&A_2 = B_G|permute([B_1, B_2]) \\
&B_1 = C_Q : (permute([C_1, C_2])) \\
&A_3 = D_G|[D_1]
\end{aligned}
$$

This gives us 24 possible paths through the post-condition. However some of these paths will result in identical results, while others are guarded, e.g: B_G is a guard to the paths $permute([B_1, B_2])$. It is possible to reduce the size of the possible paths by analysis of the post-condition - discussion of this will not be undertaken here.

3.2. Evaluating the OCL Statements

Once we have identified the various paths through the post-condition we can now generate the after-states for each path in turn. For example, one path through the post-condition may be:

$$[A_1,\ B_G|(C_Q : (C_1, C_2),\ B_2),\ D_G|D_1]$$

Using this path as our example, we can now calculate the effects on the snapshot. We have identified 5 basic operations that can be performed on the snapshot: *modify (attribute), link (two objects), unlink (two objects), create (object), delete (object)*. For each term in the list we identify a set of mappings from the OCL to these operations on a snapshot [13].

Term A_1 ($c.loan \rightarrow$ **select** $(onGoing =$ **true** $).returned =$ **true**) states that this resolves to $c1.loan$ (object: $l1$), from which those where the attribute $onGoing$ is set to true (object: $l1$). Of these the $returned$ attribute will be set to true. This produces the mapping (in this case) of $modify(l1, returned, true)$ which sets the $returned$ attribute of object $l1$ to the value **true** .

The next term is guarded by the expression B_G - if this resolves to true then the terms $(C_Q : (C_1, C_2),\ B_2)$ can be evaluated. Examination of the before-state in figure 5 shows that B_G will resolve to true. C_Q is a (existential) quantification over the set of objects $\{r1\ r2\}$. Existential quantification has a minimal obligation such that one object in the given set will satisfy the given expression, in this case $(r|r.waiting =$ **true** **and** $r.held = c)$. If there does exist an object that satisfies this then we may choose not to change any other candidate objects. In this example there are two objects which are potential candidates. The animation system may offer the user a choice over which object to continue with, however, it is envisaged that the system should be able to continue with both possibilities, including the possibility that *both* objects are modified.

When evaluating the statement after the guard B_G, expression C_1 is a modification of an attribute and behaves similarly as before, but C_2 is an assignment on a role: $r.held = c1$. In this case a link from the object r is made via role $held$ to the object $c1$ using the operation $link(held, r, c1)$. Expression B_2 is a modification of an attribute, while the guard D_G resolves to false meaning that expression D_1 is not evaluated.

This procedure is then performed over all possible execution

paths. The result of this is that from the 24 possible execution paths, there are 72 after-states generated, the greater number (24×3) coming from the fact that the quantification C_Q may generate 3 different results. Although 72 after-states are generated, there are only three unique solutions as shown in figure 7.

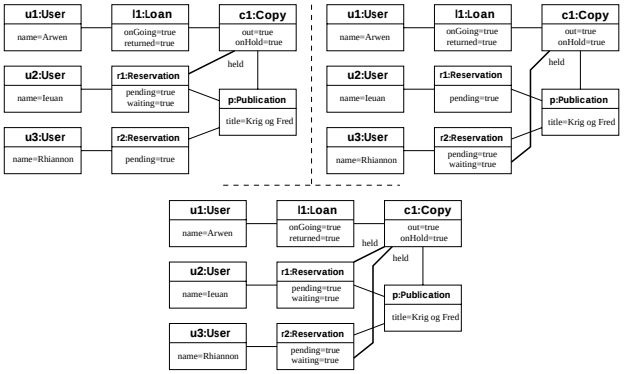

Figure 7. After-states from Animating $lib.returnCopy(c1)$

3.3. Invariants and Multiplicity Constraints

Once the resultant set of after-states have been generated, it is necessary to check those after-states against the invariants and class diagram multiplicity constraints. To satisfy some invariants[1] a change to the model may be required. We adopt a heuristic, the 'sacred post-condition,' that we may not change anything that has already been affected by the post-condition.

Each invariant is applied in turn to each of the after-states. For example if we take the object $c1$ from any of the snapshots in figure 7 we note that the attributes out, $onShelf$ and $onHold$ are $true$, $false$ and $true$ respectively. There is an invariant (see §2.1) that states that only one of these may be true at any one time. In this case the out attribute has been set to true by the post-condition and this may now not be changed which implies that all the other attributes reference in the invariant will be set to false. This process is similarly made for the other similar invariants on the Loan and Reservation classes.

The 'reservations held' invariant states that only one reservation may be held at any one time per copy. This invariant is satisfied by two of the snapshots in figure 7 but fails in the case where $c1$ is associated to both the reservation objects $r1$ and $r2$. If we try to modify the snapshot to conform to the invariant but removing one of the links we break the sacred post-condition heuristic. In this case the snapshot will never conform to the invariant and can be discarded as an invalid solution.

After applying the invariants it may be necessary to ensure that no other invariants have been affected. This can be checked by reapplying the invariants until no changes are made to the model. If under some circumstance we find that we are continually applying the same invariants then it can be assumed that some other inconsistency exists in the model as a whole. If it is found that all invariants are satisfied then the finalised after-states may be presented to the user as in figure 5.

4. Conclusion and Future Work

This paper has described a technique, using an example library system senario, where a rigorously specified object-oriented analysis model (written using UML/OCL) is animated. This animation allows the user/customer of such models to investigate the behavioural properties of the models at a much earlier stage in the modelling process than facilitated by prototyping.

We have identified the overall animation process, that is, generating the possible execution paths, evaluating each OCL term and finally applying invariants to the after-states to produce a set of 'solutions'.

There are issues such as the number of possible after-states generated, i.e: complexity issues, which we believe can be controlled by the application of suitable heuristics. For example we are investigating two heuristics we know as 'minimal obligation' and 'least interference' in which the former controls the scope of the animation of an OCL operator and the latter sorts the resultant snapshots in order of the amount of interference the operation has made to the model. We are currently working with a 'default' ordering such that linking objects is less interfering than creating intermediates and linking those as may be seen in the case of **includes** with long navigation expressions

Future work is concentrating on finalising the relationship between the OCL operators (**includes** , **exists** etc), the operations on a snapshot ($modify$, $link$ etc) and controlling the scope of an OCL operator. For example the **excludes** operator may be able to remove an object from a navigation expression by unlinking that object, deleting that object or even unlinking/deleting intervening objects.

[1] we use the term invariants here to include multiplicity constraints as well

5. Acknowledgements

Thanks to John Derrick, John Howse and Richard Mitchell for many comments, suggestions and improvements to this work as a whole. This project is supported by the EPSRC.

References

[1] J-R Abrial. *The B-Book - Assigning programs to Meanings*. Cambridge University Press, 1995. 0-521-49619-5.

[2] J C Bicarregui, D L Clutterbuck, G Ginne, H Haughton, K Lano, H Lesan, D W R M Marsh, B M Matthews, M R Moulding, A R Newton, B Bitchie, T G A Ruston, and P N Scharbach. *Formal methods into practice: case studies in the application of the B method*. IEE Proceedings in Software Engineering, 144(2):119–133, April 1997.

[3] Alex Borgida, John Mylopoulos, and Raymond Reiter. *On the Frame Problem in Procedure Specifications*. IEEE Transactions on Software Engineering, 21(10):785–797, October 1995.

[4] M Costa, J Cunningham, and J Booth. *Logical Animation*. In *Proceedings of the International Conference on Software Engineering*. 1990.

[5] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. *The IFAD VDM-SL Toolbox: A Practical Approach to Formal Specifications*. ACM SIGPLAN Notices, 29(9):77–80, September 1994.

[6] Norbert E. Fuchs. *Specifications Are (Preferably) Executable*. Software Engineering Journal, September 1992.

[7] M A Hewitt. *PiZA : User Guide*, first edition, August 1997. available from http://www.noodles.demon.co.uk/PiZA.

[8] Xiaoping Jia. *An Approach to Animating Z Specifications*. In *Proceedings of 19th Annual International Computer Software and Applications Conference*. Dallas, Texas, USA, August 1995.

[9] Cliff B Jones. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice Hall, second edition, 1990. 0-13-880733-7.

[10] B-Core(UK) Ltd. *B-Toolkit User's Manual*. B-Core(UK) Ltd., 1997.

[11] Bertrand Meyer. *Applying Design by Contract*. Computer, 25(10):40–51, October 1992.

[12] D S Neilson and I H Sørensen. *The B-Technologies : A System for Computer Aided Programming*. In *Proceedings 6th Nordic Workshop on Programming Languages*, pages 18–35. 1994.

[13] Ian Oliver. *"Executing" the OCL*. accepted at PhDOOS'99 at ECOOP'99, Lisbon, Portugal.

[14] Ian Oliver. *Validation of Object-Oriented Models using Animation*. Ph.D. thesis, University of Kent at Canterbury, 1999. in preparation.

[15] Rational Software Corporation. *UML - Object Constraint Language Specification*, version 1.1 edition, September 1997. http://www.rational.com/uml.

[16] Rational Software Corporation. *UML Notation Guide*, version 1.1 edition, September 1997. http://www.rational.com/uml.

[17] Rational Software Corporation. *UML Semantics*, version 1.1 edition, September 1997. http://www.rational.com/uml.

[18] Ian Sommerville. *Software Engineering*. Addison-Wesley, fourth edition, 1992. 0-201-56529-3.

[19] J M Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International Series in Computer Science. Prentice Hall, Hemel Hempstead, second edition, 1989. editied by C.A.R.Hoare.

[20] Margaret M West, T F Buckley, and P H Jesty. *Pelican Safety Study*. Technical Report 92.4, School of Computer Studies, University of Leeds, England, UK, March 1992.