Three Dimensional Software Modelling

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

Stuart Kent[†]
School of Computing and Math Sciences
University of Brighton
Lewes Road, Brighton, UK
Stuart.Kent@brighton.ac.uk

ABSTRACT

Traditionally, diagrams used in software systems modelling have been two dimensional (2D). This is probably because graphical notations, such as those used in object-oriented and structured systems modelling, draw upon the topological graph metaphor, which, at its basic form, receives little benefit from three dimensional (3D) rendering. This paper presents a series of 3D graphical notations demonstrating effective use of the third dimension in modelling. This is done by e.g., connecting several graphs together, or in using the Z co-ordinate to show special kinds of edges. Each notation combines several familiar 2D diagrams, which can be reproduced from 2D projections of the 3D model. 3D models are useful even in the absence of a powerful graphical workstation: even 2D stereoscopic projections can expose more information than a plain planar diagram.

Keywords

Modelling, visual formalism, object-oriented software development, CASE tools, formal methods, state charts.

1 INTRODUCTION

The advent of powerful and cheap graphical workstations, enables the use of 3D modelling in many software applications. This trend is amplified with the introduction of standards such as OpenGL [13] and VRML [3]. There are also several systems for visualising software execution, for debugging and other purposes [22], [15]. Yet, perhaps surprisingly, 3D modelling has not yet made its way into the domain of specification and design of software and information systems. Diagrams of various sorts are used extensively in this domain. These include data flow diagrams, state charts, flow charts, entity-relationship diagrams, and the diagrams in OPEN [8] and UML [21], which are all essentially 2D. It is only natural to ask: How can the introduction of 3D contribute to software and systems modelling?

This paper presents an alternative to current 2D diagrams in the form of a series of 3D graphical notations. It should be emphasised at the outset that the third dimension is not a mere embellishment. The semantics it carries considerably enhances the expressive power of graphical notations, without resorting to the clutter of textual annotations.

The notations presented here complement and are compatible with existing 2D notations such as those making up UML. In fact, each of our 3D models can be thought of as a seamless combination of several 2D diagrams. Conversely, the 2D diagrams can be extracted from the 3D model by its geometric projection onto a plane.

3D software models enable visualisation of richer semantics than those of 2D models. For example, there are 3D models that encompass both a static and a dynamic description of a system. Although we envisage a set of 3D CASE tools which support the creation, editing and manipulation of models, 3D models are useful even in the absence of such tools. A stereoscopic projection of such a model is semantically richer than a 2D diagram, since the 3D visual cues that the projection delivers, also carry a semantic meaning.

Section 2 explains why many of the notations in current use are inherently 2D and elaborates on the rationale and techniques of 3D modelling. Section 3 reviews a number of 2D notations, mostly taken from UML, the latest standard in graphical languages for the modelling of software systems. Section 4 reviews *constraint diagrams* [14], a newly proposed 2D notation, which plays a pivotal role in our 3D models. Section 5 introduces the 3D notations: contract boxes, 3D sequence diagrams, nested box diagrams, and a 3D extension of flow charts. The relationship between the 3D and 2D notations is also discussed, using the metaphor of geometric projection. Finally, Section 6 concludes the paper with a summary and outline of future work, focusing on semantics and tool support.

2 FROM 2D TO 3D MODELS

The immediate reason that diagrams used for software modelling are 2D rather than 3D might seem to be that drawing on paper is much easier than sculpturing. A moment's reflection will reveal that there must also be a deeper reason. After all, paper has been used extensively for depicting 3D models, and flat screens are effective for rendering 3D information, as in movies, TV, and in computer screens.

We attribute the prevalent use of 2D diagrams to the fact that most diagramming methods are based on graphs. For example, in flow charts the graph nodes are computational steps and edges represent control flow. Similarly, in E-R and class diagrams nodes are entities and edges are the

[*] Research done in part while the author was at the Technion Israel Institute of Technology.

[†] Partially supported by the UK EPSRC under grant GR/K67304.

relations between them, etc.

An important advantage of graph-based- over sequential textual description is that even a directional graph does not impose a total ordering of the nodes, i.e., the topology of relationship between them is not restricted to linear. In writing a Pascal program one is immediately faced with the dilemma of ordering, sometimes arbitrarily, the procedures, while the real relations between them are implicit. Not so when the program is described as a procedure call graph.

The two dimensions of a picture serve as a visual clue to the fact that graph node ordering is arbitrary and support the various topologies of a graph. The third dimension however does not contribute much to a non-textual description of a graph. Perhaps its only benefit is in making explicit (in non planar graphs) that edge-intersections are not edges as demonstrated in [20]. But this gain is all but lost when the 3D model is rendered on 2D media, unless very sophisticated rendering techniques are employed [18].

Diagramming techniques also use various generalisations of graphs, such as hyper-graphs and hi-graphs [10]. Again, these mathematical creatures are not visualised significantly better by a 3D rather than a 2D model.

As expressively powerful as graphs are, and as developed as the mathematical theory behind them is, the complexity of modelling large systems often requires more than vanilla graphs. The most common extension is that there are a number of different kinds of nodes and edges. Booch's original *Object Diagram* [2], for example, has 7 kinds of edges; his *Module Diagram* has 10 kinds of nodes.

Evidently, there is little difficulty in finding a visual, iconic representation for many different kinds of nodes. Flow charts and electrical engineering circuits use tens if not hundreds of node shapes.

The simple observation that prompted this paper is that it is immensely difficult to an adequate visual representation of different kinds of edges, due to their one-dimensional (1D) nature. Depiction techniques such as varying line thickness, dashing, doubling etc. are much more limited than the infinite variety available in 2D shape. Additionally, all of these techniques are applicable in drawing a 2D shape border. A 2D shape also leaves plenty of room for placing labels, multiple connectors and other visual adornments. In comparison, edges are limited to at most one label and to using various kinds of arrow heads and other kinds of connectors at each end. As convincingly argued in [8] connectors are inconspicuous; the viewer is forced to trace an edge to its end in order to determine its kind.

The *edge kind denotation problem* becomes acute with the multitude of relationships in object oriented systems. Class and generic instantiation, inheritance, implementation, sub-typing, ownership, inclusion, association, invocation, creation, passing as argument, state transition are only a few of the many kinds of relationships that are modelled. To overcome this problem, several kinds of graphs are often drawn to represent different aspects of the system. Nodes in these are sometimes shared and sometimes not, but almost invariably each of these graphs assigns a different meaning to edges. For example, objects in a collaboration diagram may also appear in an object-class diagram, but edges in each of these diagrams have totally different meaning. This approach ameliorates the problem by enabling an overload of edge notations. But, the relationships between different diagrams are not captured by graph theory. It is probably a consequence of this that the overall semantics of multiple diagrams is known to be one of the weakest points of modelling languages.[*]

Our thesis is that the third dimension is useful for presenting a variety of edges. The basic denotation technique we employ in addition to the usual ones is that edges with a Z co-ordinate component are of a different kind than those which remain in an XY plane. Thus, it is possible to connect together graphs of different kinds into a single 3D model that might be viewed with a 3D browser. However, as demonstrated in the sequel, a stereoscopic projection leaves enough spatial information to make the presence of a Z component of an edge easily discernible.

The technique is not limited to connecting diagrams of different kinds. By connecting two constraint diagrams, each residing in its own plane, we create a 3D model describing the pre- and post-conditions of an action (see Section 5.1). Edges residing entirely in the planes correspond to associations between the participants before and after the action. Edges traversing planes represent life-lines; they show how the participants change their state by the action.

A more advanced technique is to utilise the extra degree of freedom in drawing 1D edges in 3D. This is exemplified in Section 5 by a "lightning bolt" line to denote a procedure call edge. Helix and other kinds of edge shaping in 3D are also possible, as long as a human viewer can easily construct the 3D image from a 2D projection relying on his everyday familiarity with 2D rendering of 3D objects.

The border of a 2D shape is also 1D; this becomes an issue when it is necessary to depict the semantics of a port of connection between an edge and a node, rather than the semantics of the node or the edge itself. In 2D modelling, it is common to allocate an area in, or outside the shape to represent this semantics. The alternative 3D modelling technique we employ is that each of the *geometrical* edges defining a 2D shape becomes a face of a 3D body. If at most one *graph* edge emanates from the face, then, a 2D model drawn on the face can represent the semantics of the port. The resulting model makes more efficient use of space, but this comes at the price of a greater difficulty in 2D rendering.

Using 3D to represent different kinds of relationships by

[*] Splitting a graph to several pages, zooming and other techniques for condensing the display of large graphs do not pose a semantic difficulty. The danger sign of sloppy semantics is a model assembled from graphs which are foreign to each other.

itself may not be a new idea: Harel [10], struggling with the inability of hi-graphs to depict both set inclusion and set membership, raises a preliminary suggestion of using 3D models to overcome it, noting that

> *"Visual formalisms that are predominantly two dimensional in nature, but make some use of the third dimension, are far from being out of the question, even if we are not willing to wait for quality holographic workstations to show up."*

but without exploring the idea any further. Interestingly, the specific problem he refers to, was solved in 2D constraint diagrams. Building on these significantly contributes to the expressive power of our 3D model.

Curiously, although 3D models have been used at least twice in a taxonomy ([17] and [7]) of visual programming languages, they have found very few applications in these languages. Some exceptions though are Zhang and Zhang [23] work on distributed systems modelling in which edges traversing planes have (sometimes) different semantics and Koike's [15] Version-Module in which edges with a Z co-ordinate denote versioning, and edges without it denote assembly of a module from its components.

Below we use these techniques of 3D modelling in a series of 3D diagrammatic notations building on a small set of modern 2D notations (mostly from UML). Some readers would probably find these useful as-is, while others' visual taste in selecting symbols and shapes might differ from ours. Further research and, more importantly, experiment and experience are required before a close to an ideal visual modelling language is found. However, beyond the extra visual expressive power that our notations offer to the software engineer, the techniques should inspire further creative developments in the field of visual formalisms.

3 BRIEF REVIEW OF UML

UML is a series of 2D diagrams for modelling software and business systems. It is fast becoming the accepted standard for modelling, and it is likely that it will emerge as the official standard under the auspices of the OMG.

There is also an accompanying, precise textual language under development for annotating UML diagrams with constraints on a model that can not be expressed using the diagrammatic notation alone. This is similar to languages already proposed in the Syntropy [4] and Catalysis [5] methods.

As space is limited, we do not attempt to introduce all the diagrams and textual annotation language in this section. Instead, we only describe and give examples of diagrams that are required for purposes of comparison when the 3D notations are introduced. For a good introduction to UML, the reader is referred to [9].

3.1 Class Diagrams

Figure 1 shows the class diagram* of a toy library system which serves as a running example in this paper. The

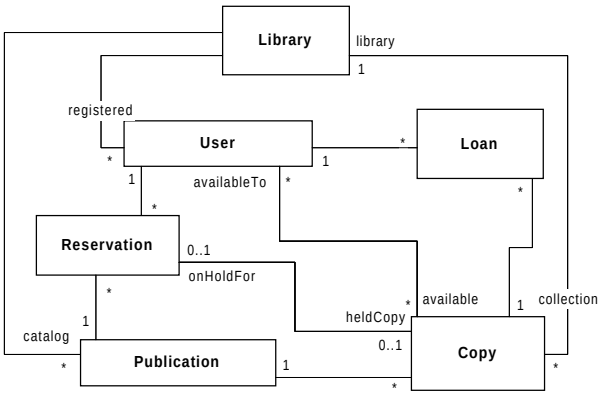


Figure 1 - UML Class (Type) Diagram

boxes represent types and the edges, associations. The meaning of the diagram is best explained in terms of the invariant constraints it places on the set of possible object configurations the model may enter.

An object configuration of a model is a collection of objects, connected by labelled links. This may be visualised using object diagrams (see Section 3.3). The type diagram says that only objects and links, of the types and with the labels appearing in the type 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 Copy and Loan indicates that a Copy object may be associated with many Loan objects, but that a Loan object must be associated with exactly one Copy object.

3.2 State Diagrams

State diagrams in UML are based on Harel statecharts [11]. They are used to model (in part) dynamic behaviour, specifically how the (abstract) state of the system changes as it responds to events.

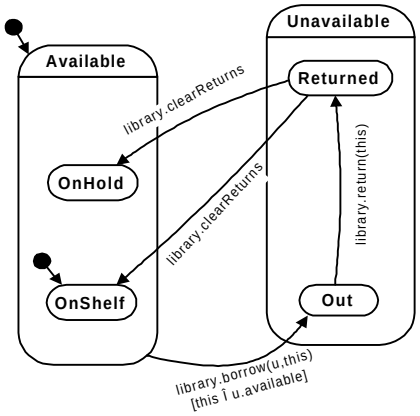


Figure 2 - UML State Diagram for Copy

* In fact "class diagram" is a misnomer, because, as [9, pp55-6] points out, in all but implementation models, there are no classes only types (interfaces). All examples used in this paper refer to *specification models*, or *design models*, hence we will only talk about types.

An example state diagram is given in Figure 2, showing the states of an arbitrary Copy object, and how that object responds to various events.

In this case the events are operations on the library system known to 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 an a shelf.

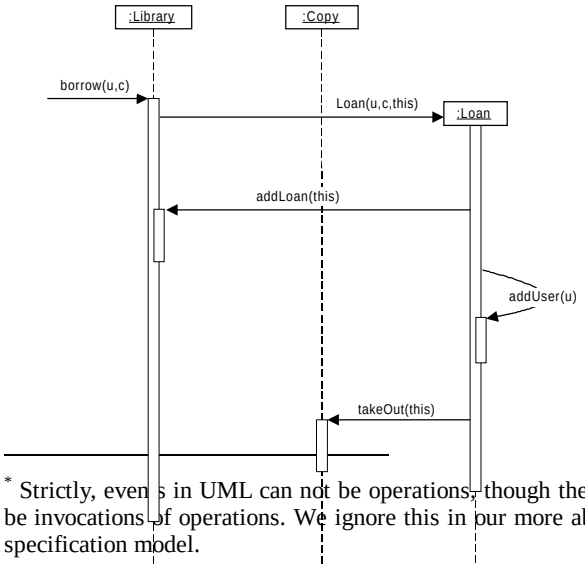
Transitions run between states. They mean that when the labelled event happens, the object changes state from the source of the transition to the target. Thus if the library associated with the copy performs a return operation and the argument to that operation is this object, then the copy will move from the Out to the Returned state. When the library performs the clearReturns operation 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. Undeterministic transitions, such as for ClearReturns, are tolerated in a specification model.

The borrow event is also associated with a guard of the form [expression]. The expression stipulates under what circumstances the event triggers the transition. In this case, the transition can not occur, if the copy is not available for lending to the user u.

3.3 Sequence, Collaboration and Object Diagrams

Sequence and collaboration diagrams are examples of *interaction* diagrams. The purpose of an interaction diagram is to show how groups of objects work together to perform a task. An object diagram can be thought of as a special case of a collaboration diagram

Figure 3 gives an example of a *sequence diagram*. Time runs down the diagram. Each object is represented by a vertical dashed line, the object lifeline. The blocks on an object lifeline represent the occurrence of an operation. The length of the block indicates the duration of the operation relative to other operations, and the positioning, the (relative) time at which the operation occurs. Arrows represent the invocation of an operation by the object at the source on the object at the target. The place where an object is introduced marks the time it is created.



* Strictly, events in UML can not be operations, though they may be invocations of operations. We ignore this in our more abstract specification model.

Figure 3 - UML Sequence Diagram

Thus the diagram shows a design for the operation borrow on the library object, which lasts the full length of the diagram. When the borrow operation is invoked, it creates a new Loan object (we use Java/C++ syntax for constructors). The creation routine in turn invokes operations on other objects, including a callback to the library object (addLoan) and an operation on itself (addUser).

The sequence diagram doesn't give a visual indication of the fact that the copy passed as an argument to the Loan, is the same copy which is sent a message by the loan later. One just surmises from the diagram that it is. Similarly, it doesn't show whether there are any existing relationships (associations) between the objects involved. A *collaboration diagram* tries to solve this problem by showing message passes and links between objects in the same diagram. Figure 4 shows Figure 3 as a collaboration diagram.

In a collaboration diagram the relationships between objects are shown, at the cost of losing a visual representation of time. Messages are shown as arrows from the source object to the target along the link used to send the message. Timing of messages is achieved by a Dewey-like numbering scheme. Some of the link labels have a «parameter» or «local» prefix, indicating that the link between the objects is by virtue of a parameter or local variable, so is transitory, rather than by virtue of a permanent association. Some of the links have two labels, indicating there are two routes between the linked objects. For example, the library knows about the copy both as one of those in its collection and via the parameter c.

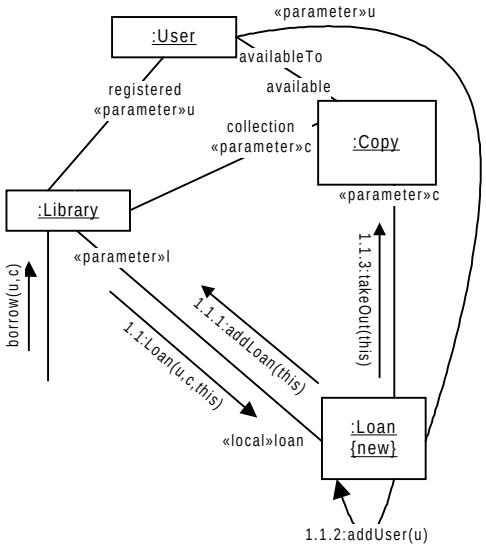


Figure 4 - UML Collaboration Diagram

A collaboration diagram is useful if the object structure remains reasonably static. However, if the structure is dynamic, that is when links and objects are continually

created and destroyed, a collaboration diagram completely ineffective. Various techniques to get around this problem have been suggested including textual annotations (used here to show that a Loan object is created), colour coding message passes and rendering their effects in the same colour, and animation in a CASE tool, where changes are shown as time progresses. None are ideal: textual annotations are not visually appealing and require much decoding of labels, there is only a limited number of clearly distinguishable colours, and animation is transient - it loses the "whole picture".

The diagrams are also not good at showing conditional statements and looping, as emphasised by [9]:

> *"One of the principal features of either form of interaction diagram is its simplicity … if you try to represent something other than a single sequential process without much conditional or looping behaviour, the technique begins to break down."*

The edge denotation problem is the reason for this difficulty. A proper visual representation of operations with control structures such as loops or parallelism requires two fundamentally different kinds of edges: one to indicate the different execution paths the operation may take (similar to flow charts) and the other to show the inter-relationships between the participants. Collaboration diagrams only show the relationships. Sequence diagrams which concentrate on time progression, make a less efficient use of 2D in using the X co-ordinate to denote objects, and therefore cannot show the potential paths of control.

In Section 5 we show how the third dimension may be used to solve many of these problems.

An *object diagram* is simply a collaboration diagram with the message passing and parameter links removed. It gives a *snapshot* of (a part of) the system state at a point in time. It is useful for illustrating states that the system may and may not be in, which can be helpful in deriving invariants, including constraints on associations. Snapshots can also be strung together in a sequence to form a *filmstrip*, to show how the state of the system changes through time. This technique is used extensively in the UML compliant Catalysis method [5].

3.4 Activity Diagrams

An activity diagram is a form of flowchart extended with notation to show processes working in parallel. It complements sequence and collaboration diagrams, as it is able to show iteration and choice, both of which are very hard to show on those diagrams. On the other hand, it is difficult to depict a message passing between objects using an activity diagram. The diagrams are similar enough to flowcharts to make an example unnecessary.

4 CONSTRAINT DIAGRAMS

[14] introduces a new notation, called *constraint diagrams*, which derives its name from its ability to express a wider range of sophisticated constraints on a model than is possible with diagrammatic notations used for object-oriented and structured systems modelling. Constraint diagrams can be thought of as an application of Harel's hi-graphs [10] in the domain of specification of object-oriented systems, with several important extensions, the main ones being the ability to show set membership, and the ability to quantify both universally and existentially over set members, in addition to set plain containment. This is achieved in part by introducing notation for showing singletons (and sets of other cardinalities).

The notation is summarised here, using the example in Figure 5 for illustration.

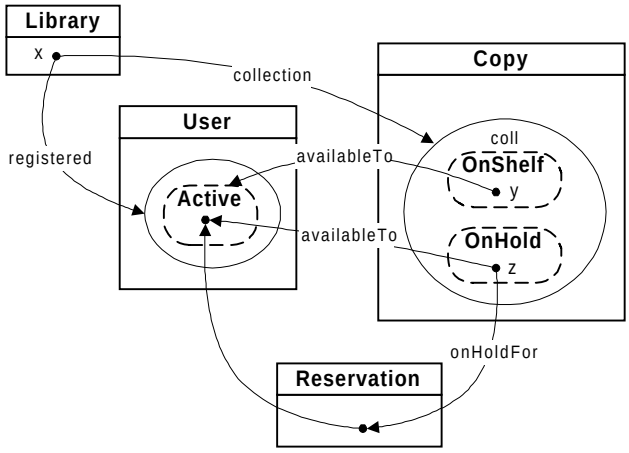


Figure 5 - Constraint Diagram

Figure 5 documents an invariant which can not be expressed using the diagrammatic notations of UML, even enhanced with ideas from other OO notations such as [6]. The only other way to express these precisely is through the use of a textual, mathematical language, such as that suggested for [4], [5], or [19]

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 had been made, then it is only available to the registered user that made the reservation.*

Let us 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 2 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 is the link, which corresponds to associations in type diagrams.

The starting point for reading Figure 5 is x. This is an arbitrary singleton set (i.e., object) of the type Library. It is the starting point because it is the *least defined set* on the

diagram*. The fact that it is a singleton is indicated by a small filled circle. A set with zero or one elements is shown by ○, and an empty set by ≐ (null pointer) or ⬭. Other sets may have any number of elements, including zero, unless an explicit numerical expression (e.g., ≥ 2 & ≤ 10) indicating otherwise is attached as a label.

x is the least defined set because it is universally quantified (it has no incoming arrows, is not a state, type, an intersection or existentially quantified), and the set in which it is contained (a type) is defined by default. In contrast, the sets y and z are also universally quantified, but are more defined than x because they are contained in sets which themselves are partly defined by the collection link from x. Specifically 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 a 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.

A formal rendering of this invariant could be derived in a similar way.

The full notation also includes symbols for showing different set cardinalities, inheritance relationships between types, nesting of states and existentially quantified elements. The reader is referred to [14] for the details.

* In general, there can be more than one least defined set, in which case there is a choice of where to start.

5 3D MODELLING NOTATIONS

This section describes the 3D notations we have developed so far: contract boxes, 3D sequence diagrams, nested and structured box diagrams. Also discussed are the relationships between these diagrams and the UML diagrams introduced in Section 3. The metaphor of geometric projection is used to explain how the latter may be retrieved from the former.

5.1 Contract Boxes

In methods such as Catalysis and BON [19], a precise mathematical language is provided for placing static and dynamic constraints on a model, that can not be expressed diagrammatically in languages such as UML. As we have seen above, constraint diagrams may be used to characterise static, invariant constraints. Contract boxes allow constraints on dynamic behaviour to be expressed that would otherwise require textually expressed pre/post contracts (as in traditional formal methods, see e.g., [12]) on operations.

The notation is actually quite simple. It can be thought of as a box, hence the name contract box, with a constraint diagram pasted to the lid and another pasted to (the inside of) the bottom, as illustrated by Figure 6. The constraint

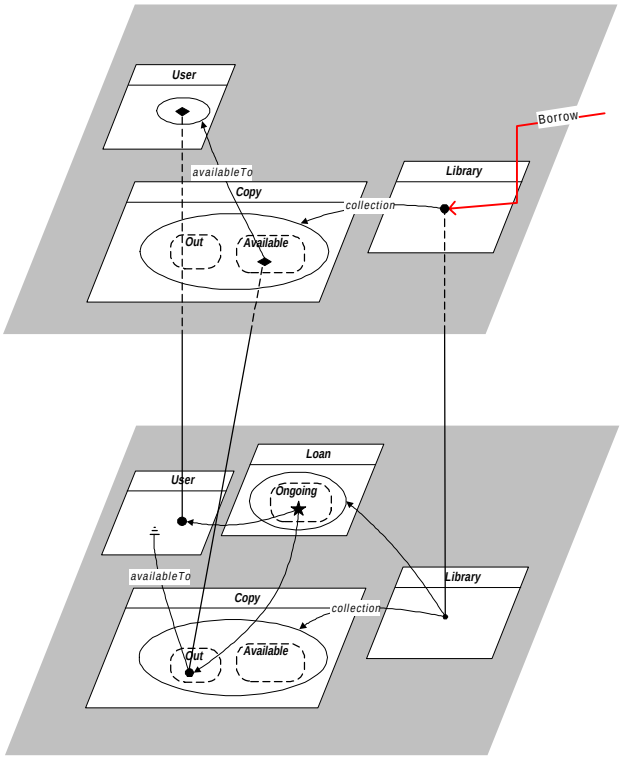


Figure 6 - Contract Box

diagram on the top constrains the pre state, and the one on the bottom the post state. Thus the top diagram contributes both to the pre-condition and part of the post condition (which is a predicate of both states), and the bottom diagram contributes only to the post condition. If desired, the top diagram could be thought of as a layering of two

diagrams - one for the pre-condition and one for the post condition. Making this layering explicit might be necessary for complex specifications.

The constraint diagram notation is extended slightly with a special symbol for input parameters of the operation. In this stereoscopic projection a diamond symbol has been used*.

In Figure 6, two parameters of the actions are shown, a copy and a user. The pre-condition constraint represented by this diagram is that the copy, which must be an available one in the collection of the library on which the operation is being invoked, must be available to a set of users including the parameter to the operation, which in this model is the one borrowing the copy.

The object on which the operation is being invoked is shown by a message send which is depicted by a "lightning bolt" to that object, labelled with operation being invoked.

The post-state diagram shows the change in state that the operation has caused. Object *lifelines* are used to show that an object taking part in the post-state is the same as another depicted in the pre-state. In general, lifelines can be used to connect sets. Here, the effect of the operation is to:

- change the state of the input copy object to Out, which is shown by moving the object from the projection of the Available state to the projection of the Out state;
- ensure that the set of objects the copy is availableTo is empty ($\doteq$);
- create a new loan object in the OnGoing state (indicated by a ★); and
- associate the loan object with the user and the copy.

This illustrates a further extension to the constraint diagram notation, specifically to indicate the creation of objects. The creation of a single object is indicated through the use of a ★, of a set with 0 or 1 objects by ☆, and of a set with an arbitrary number of objects by ☆.

Some of this behaviour, namely the pre condition that the copy object must be in the available state and the fact that it changes state during the operation could be shown by a state diagram in UML. Indeed a projection of this diagram would be a part of the state diagram for Copy given in Figure 2. It would be constructed as follows. Focus on the copy object that changes state (if there was more than one, then repeat the process for each). Draw a state diagram with a single transition labelled library.borrow(u.this) (the operation being performed is borrow on the library object obtained by navigating backwards down the collection link from the copy object under consideration). The source of the transition is the state in which the copy object appears in the pre-state diagram, and the target is the state in which it appears in the post-state diagram. Finally, place a guard on the transition to the effect that the copy (this) must be available for lending to u.

A textual (informal or formal) version of the contract could also be derived from the diagram in a similar way to the derivation of invariants from constraint diagrams.

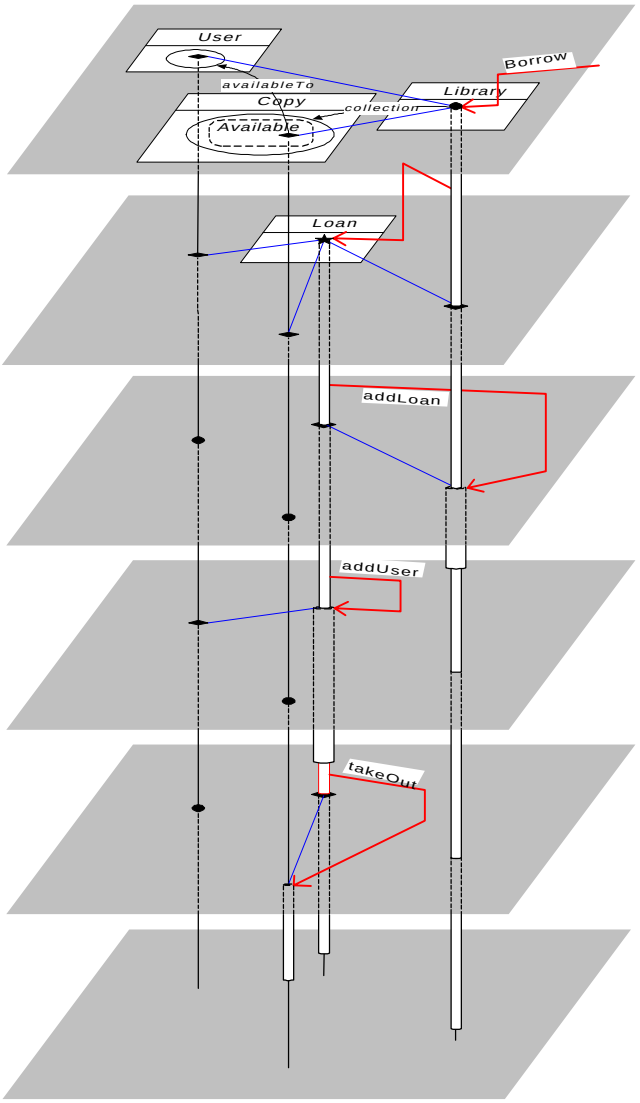


Figure 7 - 3D sequence diagram

5.2 3D Sequence Diagrams

This section shows how the third dimension can usefully be employed to provide much richer visualisations of action designs, than is possible with UML sequence and collaboration diagrams. Indeed, we contend that both of these kinds of diagram are projections of a single 3D sequence diagram.

Figure 7 gives an example of the 3D sequence diagram for the borrow action. Each plane represents the state of the system at which an object is sent a message. As before, lightning bolts represent message sends. A constraint diagram may be used at each plane to show the context of input parameters, and, if desired, pre-condition

* In a true 3D rendering, we envisage using "pits" in the top plane for input parameters, and "pegs" sticking out of the bottom plane for output parameters.

information. Thus the top plane in Figure 7 shows the parameters of the operation borrow, invoked on the library object together with their context, which happens in this case to also correspond to the pre-condition of that action. In subsequent planes, that context need not be repeated, as it doesn't really change. Parameters have been shown both by the diamond notation and by lines linking them to the lifeline of the object on which the operation is invoked. The latter helps to highlight which objects, or sets of objects, are involved in the operation being invoked. They can be included as there is little risk of "edge clutter".

Lifelines trace the life of an object, or set of objects, through time. The length of actions is shown by cylinders running down the object lifelines. Cylinders may be wrapped in the case of callbacks. Thus there is a cylinder running the full length of the diagram down the library object lifeline corresponding to the borrow operation. The third message pass (addLoan) is a callback to the library object and is shown as cylinder wrapping that for borrow.

If necessary an additional plane may be inserted where an action completes, to show output parameters and their context. However, this should be done only where necessary. If that level of detail is regularly required, then a nested box diagram (see below) should be used instead.

A 2D sequence diagram is clearly a projection of the richer 3D variant, as exemplified by this and Figure 3. In the 2D version, object lifelines and operation cylinders are mapped into lifelines and operation blocks attached to them, respectively. The lightning bolts are mapped into message sends. The main information that is lost is about the players in the game, which is here provided by the constraint diagrams in the horizontal plane. However, in a 2D sequence diagram, copious labelling of lifelines and action parameters is required to show that a message send is to an object that has previously been passed as an operation parameter, resulting in a cluttered and clumsy representation Not so here, where it is also possible to show information about the context of each parameter, for example that the copy is assumed available to the user and in the library's collection.

In UML some of this contextual information is instead provided by collaboration diagrams, though at the cost of losing the visual depiction of time flow. However, even in collaboration diagrams it is still necessary to use labelling to identify operation parameters with objects appearing in the diagram.

An alternative projection would be to collapse the diagram on the time dimension, using the cylinder information to devise a scheme for numbering message sends. The result would be an extended form of UML collaboration diagram; extended because a constraint diagram would be used to characterise the static configuration of the players in the game, in a much more general way than is possible with an object diagram.

In conclusion, the 3D sequence diagram brings together both the sequence and collaboration diagrams of UML, into a single diagram, whilst retaining the advantages of both. In addition it carries more information about the static configuration of the major players, by its use of constraint diagrams in place of object diagrams.

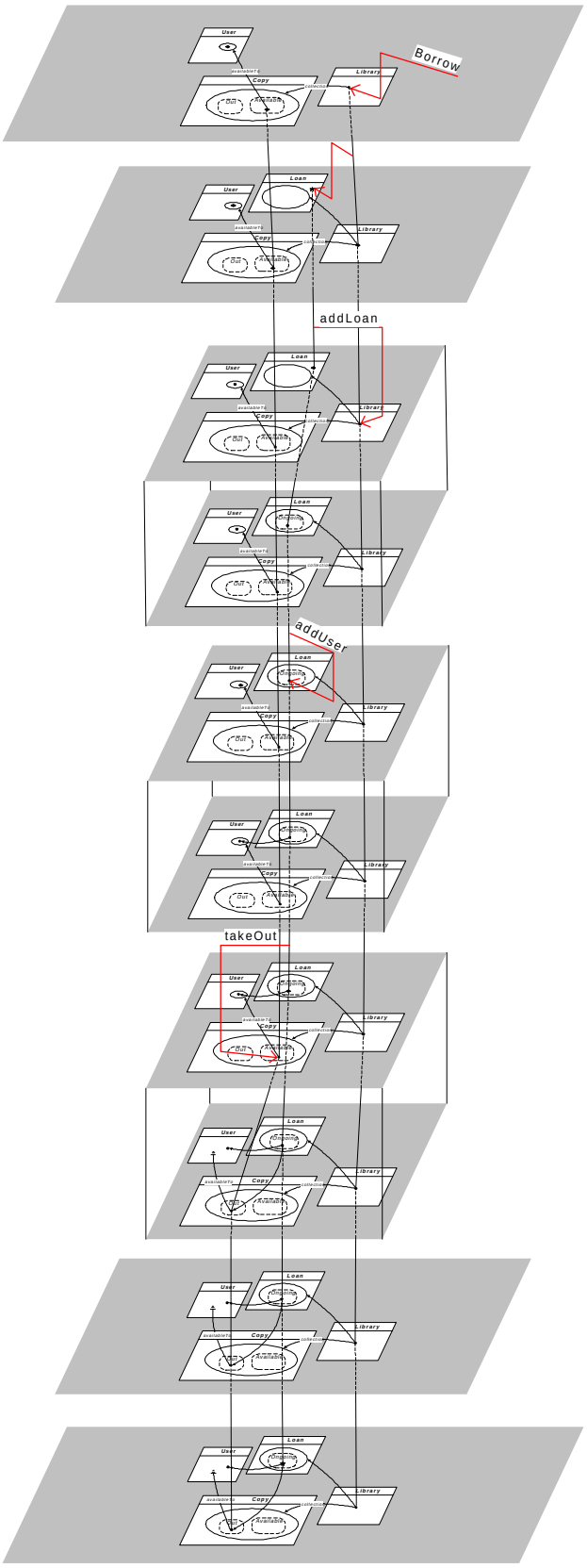


Figure 8 - Nested Box Diagram

5.3 Nested Box Diagram

The 3D sequence diagram shows message passes between objects and sets of objects, and allows the various players in the game to be easily identified. It shies away from actually specifying the behaviours of the actions involved. Of course these could be specified using a series of contract boxes in separate diagrams. However, it is also possible to combine the contract box specifications with the sequence diagram resulting in a visualisation of the complete action design, called *nested box diagram*, which includes the specifications of the sub-actions involved. Figure 8 shows this for the borrow action.*

Each box corresponds to an action. Nested operation invocations are shown by nesting boxes, so cylinders down object lifelines are no longer required. Furthermore, the full detail of the behaviour of each operation is shown on the diagram, as each box characterises an operation contract, describing the behaviour of that operation for this context. An edge connecting a set to a singleton parameter denotes an implicit loop of calls for each set member. This convention minimises the need for denoting control structure.

However, including this much detail on the diagram makes it hard to read. The problem would be exacerbated if deep nesting of boxes was required. Nevertheless, we can imagine a sophisticated visualisation tool which would allow one to view such a diagram at different levels of detail, zooming in on certain features, opening up boxes to see the those nested inside it, and so on.

Another way of looking at this diagram is as a visual representation of the underlying semantic model that ties the various perspectives on the specification and design of the borrow operation together.

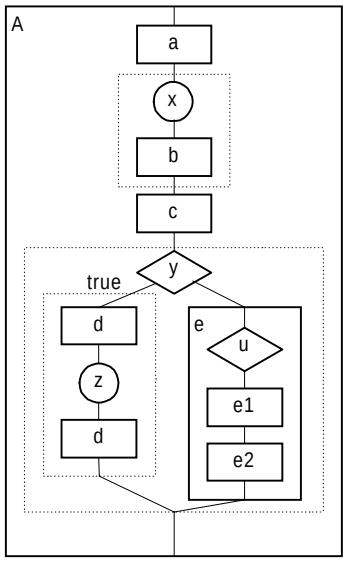
Nested box diagrams provide the specifications of all operations used in the design, and of the borrow operation itself. State diagrams could be projected from these boxes as described in Section 5.1. A 3D sequence diagram could be projected from these to show more clearly the message passing between objects. UML sequence and collaboration diagrams could, in turn, be projected from this (the 3D diagram).

What is missing is a fuller specification of the static behaviour: types, associations, invariants and states. These could be included by layering the top and bottom of each box, as suggested in Section 5.1. A single constraint diagram in Figure 8 would be replaced by a stack of constraint diagrams, some corresponding to pre/post conditions and some corresponding to invariants, noting that type diagrams may be mapped into constraint diagrams, and states are already included on constraint diagrams.

5.4 Structured Box Diagrams

Structured Box Diagrams can be thought of as a 3D generalisation of flow charts of *block structured* programs (or UML activity diagrams). They are also an extension of nested box diagrams. In such flow charts, program-block nesting could be read from the chart if the graph layout obeys some simple rules. A visual representation of block nesting is also given by the famous Nassi-Schneiderman (N-S) diagrams [1], but these take a more geometric than a graph-like approach.

Figure 9 demonstrates one possible hybrid alternative in which the steps of a flow chart can be grouped together in rectangles to denote nesting. Dotted rectangles denote anonymous program blocks, while a solid named rectangle, andrectangle containment corresponds to block nesting. Figure 9 (b) gives the pseudo-code semantics of the chart in Figure 9 (a). The other conventions used are simple: a circled condition asserts the next rectangle is executed in a 'while' loop guarded by that condition; a condition in a lozenge indicates that the following rectangle is executed at most once contingent on that condition, (as in condition u and operation e1). A choice between two (or more) options is denoted by a lozenge with several multiple exits. All exits from a lozenge except for one must be labelled. An exit must lead to a single rectangle, and the exit points of all these rectangles must merge. If the branch does not consist of a single operation then an anonymous block is introduced, which is denoted by a dashed rectangle. All other dashed rectangles are strictly optional.



(a)

```
Operation A
    a;
    while x do b;
    c;
    if y then begin
        d1;
        while z do d2;
    end else e;
end A;

Operation e
    if u then e1;
    e2;
end e;
```

(b)

Figure 9 - Structured Flow Chart

By adding a third dimension to a nested rectangle chart we obtain a *structured box diagram*. This is done by replacing each operation rectangle with its contract box. Circles are replaced with spheres and lozenges with tetrahedra. Thus, the nested rectangle chart remains a 2D projection of the box diagram. Constraint diagrams can also be used effectively to define conditions for choice and loop guards. These would appear inside the sphere representing a loop and on the appropriate faces of the tetrahedron

* This diagram has been reduced in size just so we can fit it on the page. At full size all the details are visible.

representing a choice.* It is also possible to extend N-S diagrams to 3-D in much the same way.

6 CONCLUSIONS AND FURTHER WORK

We have used the third dimension to illustrate a different kind of arrow in the same diagram. This has been illustrated by a series of 3D notations, which have a number of advantages over their 2D counterparts.

Contract boxes visualise behaviour that in UML can only be shown using a combination of state diagrams and precise textual language. 3D sequence diagrams visualise collaboration between objects in a way that avoids confusing labelling schemes, (e.g., of parameters), and is able to show visually both the flow of time and the key players and their context. UML requires 2 kinds of diagram to show this information. Nested box diagrams combine 3D sequence diagrams and contract boxes into one diagram. Structured box diagrams extend nested box diagrams to visualise looping and conditional behaviour, incorporating constraint diagrams to visualise conditionals and loop guards. Thus structured box diagrams show all aspects of the model on a single diagram. All other diagrams may be regarded as either sub-diagrams or projections of these.

The primary goals of future work are to refine the notation and provide tool support, with both activities informed by feedback from the use of the notations in practice.

Formal work is required to check the expressiveness and integrity of the notation, to define a calculus for composing models, (which is particularly important for component-based development), to develop techniques for checking the semantic integrity of models and their compositions, for mapping models to current OO programming languages, for defining projections from the 3D notations, and so on.

Some of this work should be directed toward the provision of CASE tools. These are essential to the use of the notation for "real-world" projects, (even stereoscopic projections take much longer to draw than simple 2D diagrams), and this is a prerequisite for obtaining informed feedback. Specifically we envisage tools to assist with model visualisation and model construction.

Visualisation. We imagine a sophisticated 3D rendering system which would allow developers to "walk through" and, possibly manipulate, the model, probably viewed as a box diagram. The system would allow the model to be seen from different perspectives, and allow developers to zoom in on and uncover more detailed aspects, for example opening up a box to reveal its implementation, which would itself be another box diagram. At its outermost level, a model would be a collection of boxes, the operations that it is able to perform. The encoding of the diagrams in VRML [vrml] could enable a prototype to be built relatively cheaply.

The visualisation tool should also allow projections of the model to be taken. That is extract the UML diagrams, 3D sequence diagrams, contract boxes etc. The static object structure of the system could be constructed, by e.g., combining the invariant constraint diagrams layered in the operation boxes. Other projections might not be visual: for example, it might be possible to construct textual specification in English legalise (e.g., [1]) from the constraint diagrams and contract boxes.

Model construction. Building the models by drawing the 3D diagrams is untenable, excepting the possibility of direct manipulation through a VR interface. Instead, it should be possible to do the reverse of projection, and build 2D and simpler 3D diagrams which are then used to construct the complete 3D model, which in turn is visualised by the tool. For example, we imagine that the constraint diagrams for a contract box would be constructed by drawing up the first diagram and then marking the changes by direct manipulation (e.g., moving an object from one state to another). This should provide enough information for the tool to render the box in 3D.

Alternatively, formal textual descriptions could be used to describe the changes - this object moved from here to there, and the tool would proceed to construct diagrams from this textual description.

A third possibility is to generate the generic characterisation of the model, e.g., type diagrams, constraint diagrams, contract boxes, state diagrams etc., from the specific, e.g., snapshots (object diagrams) and filmstrips. Kent [14] discusses this idea further.

7 REFERENCES

[1] Assertion Definition Language (ADL), The Open Group (formally X/Open), http://adl.xopen.org, 1997.

[2] Booch G., Object Oriented Design with Applications, Benjamin/Cummings, 1991.

[3] Carey R. and Gavin B. The Annotated VRML 2.0 Reference Manual, Addison Wesley 1997

[4] Cook S. and Daniels J., Designing Object Systems, Prentice Hall Object-Oriented Series, 1994.

[5] D'Souza D. and Wills A., Component-Based Development Using Catalysis, book manuscript available at http://www.iconcomp.com, 1997.

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

[7] Freeman E, Gelernter D., and Jagannathan S. In Search of a Simple Visual Vocabulary. Proceedings. of the 11th IEEE Symposium on Visual Languages, 1995.

[8] Firesmith D., The Open Modelling Language, SIGS, 1997.

[9] Fowler M. with Scott K., UML Distilled, Addison Wesley 1997.

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

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

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

[13] Kempf R. and OpenGL Architecture Review Board. OpenGL Reference Manual, Addison Wesley Longman, 2nd Edition, 1997.

[14] Kent S., Constraint Diagrams, to appear in the proceedings of the 12th Annual Conference on Object Oriented Programming Languages and Systems (OOPSLA'97), ACM Press, 1997.

[15] Koike H. The Role of Another Spatial Dimension in Software

* In the case of a simple yes/no choice the constraint diagram would appear on the central, horizontal plane of the tetrahedron.

Visualization. ACM Trans. Inf. Sys. 11(3):266-286 July 1993.

[16] Nassi I. And Schneiderman. Flowchart Techniques for Structured Programming.. ACM SIGPLAN Notices, 8(8):12-26.

[17] Shu N. C. Visual Programming. Van Nostrand Reinhold Company, 1988.

[18] Spratt L and Ambler A. Using 3D Tubes to Solve the Intersecting Lines Problem. Proceedings. of the 10th IEEE Symposium on Visual Languages, 1994.

[19] Waldén K. and Nerson J-M. Seamless Object-Oriented Software Architecture - Analysis and Design of Reliable Systems, Prentice Hall 1994

[20] Ware C. and Franck G. Viewing a Graph in a Virtual Reality Display is Three Times as Good as a 2D Diagram. Proceedings. of the 10th IEEE Symposium on Visual Languages, 1994.

[21] Unified Modeling Language (UML) v1.0, Rational Software Corporation, available at http://www.rational.com, 1997.

[22] Vion-Dury J-V and Santana M. Virtual Images: Interactive Visualization of Distributed Object Systems. Proceedings of 8th OOPSLA , 1993.

[23] Zhang D-Q and Zhang K., A Visual Programming Environment for Distributed Systems. Proceedings. of the 11th IEEE Symposium on Visual Languages, 1995.