Requirements as conjectures: intuitive DVD menu navigation

Jemima Rossmorris and Susan Stepney

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK.
susan@cs.york.ac.uk

Abstract. In this paper we use Z to capture the requirements for an 'intuitive' menu navigation system as a series of conjectures that should hold. We use those requirements to investigate potential algorithms. The Z formalisation enables the somewhat fuzzy requirement of 'being intuitive' to be captured precisely, analysed, and critiqued, leading to possibly new requirements, and more intuitive algorithms.

Keyword: Z, requirements, conjectures, DVD

1 Introduction

Interactive systems require some sort of input from the user to instruct the system on what behaviour is desired. One way of interacting is for the user to navigate around a menu system, selecting options as necessary to achieve the desired behaviour. Examples of this are navigating a TV menu to change the brightness and contrast, navigating a DVD menu to chose an episode or special feature, and navigating the links on a Web page. Normally Web page navigation is done using the mouse to control a cursor. However, blind and motion-impaired users may not be able to use a mouse, and are often restricted to using the tab key, or cursor keys, to navigate. Most TV and DVD controls are similarly restricted, and have some form of cursor that jumps between menu items.

The W3 guidelines on the use of the tab key state that web designers should "create a logical tab order through links, form controls, and objects" [WC3, checkpoint 9.4]. In practice, the order in which links are tabbed through on a web page is often taken to be the order in which the links are defined in the html code. This order may be the desired one, but it may not: it is left to the page designer to get it right.

For DVDs the situation is even worse. There is no standard or guidelines for how the cursor should react. As Donald Norman puts it: "Designers haven't figured out the cursor model yet either: In most DVDs, pushing the joystick (or arrow) control up will move the cursor up, but I have encountered some in which the cursor moves down." [Norman 2001].

Ideally, cursor navigation should be 'intuitive', that is, predictable to the average user. Yet consider figure 1 (a layout seen on real DVD menus). In the oval layout, is item 2 or item 4 to the 'right' of item 1? In the diamond layout,

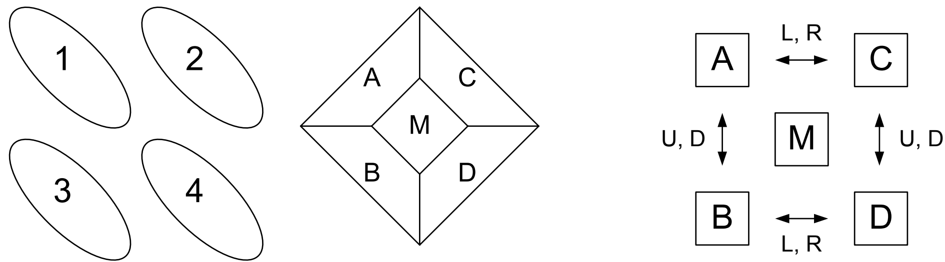

Fig. 1. Menu navigation: (a) two potentially ambiguous menu layout arrangements (taken from real DVD menus) (b) a navigation that misses the middle item

is item C or item M to the 'right' of item A? If the cursor moves to C, how does it get to M? If it moves to M, where next, to C or to D? And so on.

The cursor navigation problem is a real, non-trivial, unsolved problem in set-top box,[1] DVD, and other interactive design. It is separate from the screen design problem. Individual screen designers layout their menu items in whatever manner they see fit; the screen display provider (for example, the set-top box manufacturer) then has to provide a general purpose 'intuitive' algorithm to navigate the cursor on any potential screen layout.

One selling point of formal methods is their ability to capture abstract requirements without being distracted by implementation detail. For example, a very abstract Z [ISO-Z 2002] specification would be written as the high level requirements specification, and then refined down to include more concrete implementation details. The usual approach is to write the Z specification in a 'state and operations' style. However, requirements are not always properties of single operations. In this paper we use Z to capture the requirements for an 'intuitive' menu navigation system as a series of conjectures that we want to hold on our (initially very underspecified) state and operations specification. We use those requirements to investigate potential operation specifications. In turns out that some obvious requirements (such as 'undo') put particularly strong constraints on the design, whilst others (such as the seemingly innocent 'right arrow moves the cursor right') are in fact very difficult to capture, not at all intuitive, and bear further investigation. The Z formalisation enables the somewhat fuzzy requirement of 'being intuitive' to be captured precisely, analysed, and critiqued, leading to possibly new requirements, and more intuitive algorithms.

2 Basic specification

We start by specifying the minimum necessary to capture the requirements. We simplify the problem by considering the screen to be tiled with squares, each of which may be a menu item.

[1] C. A. Whyte, private communication

The screen is *xSize* units wide and *ySize* units tall.

$$xSize, ySize : \mathbb{N}_1$$

Positions on the screen are given by a pair of (x, y) coordinates, each numbered from zero up to the maximum size. x coordinates increase in the rightwards direction; y coordinates increase in the downwards direction. We require there to be at least two screen positions, in order to exclude trivial menus.

$$position == 0 \mathrel{..} (xSize - 1) \times 0 \mathrel{..} (ySize - 1)$$

$$1 < \# position = xSize * ySize$$

The screen itself is then a set of (at least two) *menu* positions, and a *cursor* positioned on one of the menu items.

$$\begin{array}{l} \textit{Screen} \\ \hline menu : \mathbb{F}\, position \\ cursor : position \\ \hline 1 < \# menu \\ cursor \in menu \end{array}$$

The basic cursor movement does not change the *menu* items. (We do not provide any operations to change the menu items: it is assumed the screen is initialised with the desired items.)

$$BasicMove == [\, \Delta Screen \mid menu' = menu \,]$$

The *MoveRight* operation moves the cursor 'rightwards'. We do not yet specify what that means, but we need to provide the declaration for use in the requirements conjectures. We can consider this specification to be parameterised by the *MoveRightPredicate*, which needs to be chosen such that it fulfils the requirements.

$$MoveRight == [\, BasicMove \mid MoveRightPredicate \,]$$

Similar declarations are made for the other cursor directions. Then the general *Move* operation is a movement in one of the cursor directions.

$$Move == MoveRight \lor MoveLeft \lor MoveUp \lor MoveDown$$

3 Requirements as conjectures

We now have enough machinery to capture the requirements. There are several requirements that the navigation system should 'clearly' fulfil. We analyse the consequences in the following section. We focus on *MoveRight*: the other directions follow by symmetry arguments.

3.1 Conjectures in Z

In ISO Standard Z [ISO-Z 2002], a conjecture paragraph has the following syntax:

$\vdash ?$ `Predicate`

In a well-formed specification, the `Predicate` must be well-typed in the context in which it appears, but it need not be *true*. If it is *true* (is implied by the properties of the specification) then it is said to be *valid*, and is then a *theorem* of the specification. (See [Valentine *et al.* 2004, chapter 5] for further explanation.)

In this paper, we use the following extension to the standard syntax in order to separate out declarations from the body of the conjecture:

`SchemaText` $\vdash ?$ `Predicate`

is equivalent to

$\vdash ?\ \forall$ `SchemaText` $\bullet$ `Predicate`

3.2 R1. Is Deterministic

The cursor's movement should be consistent with its historical behaviour. That is, whenever the cursor is on a particular menu item and you press a particular arrow key, the cursor should always move to the same next menu item.

This is captured by requiring each separate part of the *Move* operation to be deterministic, or functional.

$$\vdash ?\ \{\ MoveRight \bullet cursor \mapsto cursor'\ \} \in position \nrightarrow position$$

3.3 R2. All Reachable

Whichever menu item you start from, you should be able to reach any other menu item by a sequence of arrow key presses. (This might seem obvious, but there are web pages using frames where this does not hold, as it is not possible to tab between their frames.)

$$Screen \vdash ?\ \{\ Move \bullet cursor \mapsto cursor'\ \}^* = menu \times menu$$

3.4 R3. Undo

Being able to undo one's actions is an important feature in any interactive system [Abowd & Dix 1992]. Undo in navigation is useful if you go past the link you want: you can go backwards, instead of having to scroll through all the links again.

So *MoveRight* followed by *MoveLeft* should leave you back where you started. (It is possible to consider the use of a 'shift' key, giving a separate *MoveLeft* and 'undo *MoveRight*', but that adds complexity to the user interface.)

$$MoveRight \fatsemi MoveLeft \vdash ?\ \Xi Screen$$

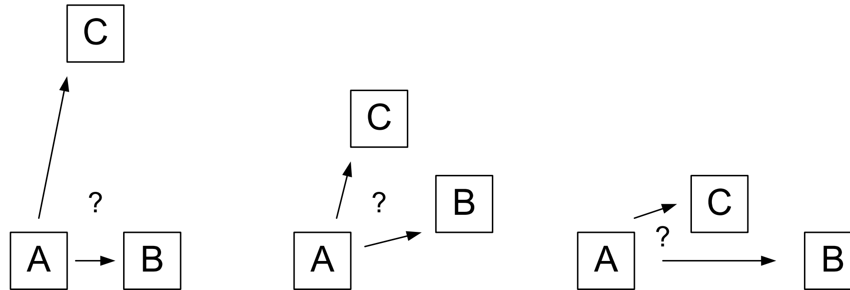

Fig. 2. What is right?

3.5 R4. Move moves

Feedback is another important feature. If you press a key, something should happen. We capture this with the requirement that, no matter where the cursor is, if you press an arrow key, the cursor moves to another menu item. (We could alternatively require it to 'flash', say, if there was nowhere for it to sensibly move.)

$$Move \vdash ?\ cursor' \neq cursor$$

3.6 R5. MoveRight moves right

The previous requirements are all at a very basic level, that pressing a key does something sensible. This requirement captures a more detailed level of the intuition, that pressing a key does the 'correct' thing. So, no matter where the cursor is, if you press the right arrow key, the cursor should move to the next menu item to the right.

This requirement is suprisingly difficult to capture, and is where all the complexity, and most of the interest, lies.

We want the cursor to move to the 'nearest' 'rightward' menu item, including wrapping around when close to the right edge of the screen. (It is much easier to specify if wrapping is not allowed, but that is considered to be a simplification too far, as real DVD menu navigation *does* wrap.)

Just what, however, is 'right' of the current position? Consider figure 2. In the lefthand pcture, should the cursor move from A to C (which has the smaller x-coordinate distance from A) or to B (which has the smaller Euclidean distance from A)? One feels that 'intuitively', the choice should be B. In the middle picture, B and C have the same Euclidean distance from A, and C has the smaller x-coordinate distance. One again feels that intuitively, the choice should be B, (although an argument might be made for C). B seems to be more 'rightward', and C more 'upward', as can be seen from the angle of the arrows pointing to them. However, it is not just the angle of the movement. In the righthand picture, B is definitely more rightward in terms of angle, yet in this case, C seems to be

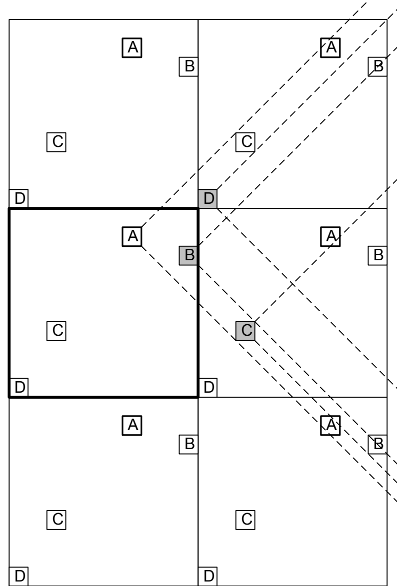

Fig. 3. right-cones in the tiled plane, with the original screen shown in bold

the intuitive 'right' choice. There is some combination of angle and distance. This is the intuition captured by the following requirement (which first needs a little machinery to be defined before it can be stated succinctly).

The definition is complicated by requiring wrapping when the cursor is at the far right. The usual way to model this is by modulo arithmetic. However, here it will be easier to visualise the effect by the isomorphic 'infinite tiling' approach. The entire plane is tiled with copies of the *Screen* rectangle, with corresponding menu positions and cursor (see figure 3). So each screen *position* is mapped to a corresponding set of tiled points in the plane:

$$tile == \lambda\, p : Point \bullet \{\ i, j : \mathbb{Z} \bullet (p.1 + i * xSize, p.2 + j * ySize)\ \}$$

tile takes a *Point* (defined in the appendix), and gives the set of all *Point*s resulting from tiling the plane in $xSize * ySize$ tiles.

We then capture the difference between 'rightward' and 'upward' (or, symmetrically, 'leftward' and 'downward') directions, as a rightward viewing area subset of this tiled plane. The view to the right of a position is all those points within a right-pointing cone with 45 degree semi-angle, apex at the position of interest (see appendix for the definitions of $cos45$ and *unitVector*), as shown in figure 3.

$$viewRight == \lambda\, p : Point \bullet$$
$$\{\ q : Point \setminus \{p\} \mid cos45 \leq (unitVector(q -_v p)).1\ \}$$

Now we define a screen of menu items, wrapped into this rightward view. *rightMenu* defines all rightward menu items as all the menu items in the infinite tiled plane, restricted to those in the rightward cone. *rightMenu*1 picks out the one of these menu items in the rightward cone that is closest to the *cursor* (see appendix for the definition of the distance d_R). There may be more than one tiled menu item at the same distance from the cursor; we leave the definition loose at this stage.

WrapScreenRight

$$
\begin{array}{l}
\mathit{Screen} \\
\mathit{rightMenu} : \mathit{Point} \pfun \mathbb{P}\,\mathit{Point} \\
\mathit{rightMenu}1 : \mathit{Point} \pfun \mathit{Point} \\
\hline
\mathit{rightMenu} = \lambda\, m : \mathit{menu} \bullet \mathit{tile}\ m \cap \mathit{viewRight}\ \mathit{cursor} \\
\forall\, m : \mathit{menu} \bullet \\
\quad \mathit{rightMenu}1\ m \in \{\ p : \mathit{rightMenu}\, m \mid \\
\qquad \forall\, q : \mathit{rightMenu}\, m \bullet d_R(\mathit{cursor}, p) \leq d_R(\mathit{cursor}, q)\ \}
\end{array}
$$

Then the *nearest* menu item to the cursor, on the right, is the *rightMenu*1 menu item that is no further from the cursor than any other. (Again, this is loose if there are several at the same closest distance.)

RightView

$$
\begin{array}{l}
\mathit{WrapScreenRight} \\
\mathit{nearest} : \mathit{position} \\
\hline
\mathit{nearest} \in \mathit{menu} \setminus \{\mathit{cursor}\} \\
\forall\, m : \mathit{menu} \setminus \{\mathit{cursor}\} \bullet \\
\quad d_R(\mathit{cursor}, \mathit{rightMenu}1\ \mathit{nearest}) \leq d_R(\mathit{cursor}, \mathit{rightMenu}1\ m)
\end{array}
$$

So, finally, the requirement is that a *MoveRight* operation moves the cursor to the nearest right menu position:

$$\mathit{MoveRight};\ \mathit{RightView} \vdash ?\ \mathit{cursor}' = \mathit{nearest}$$

The corresponding requirements on the other direction operations follow by symmetry.

4 Satisfying the requirements

4.1 Analysing the requirements

We now consider the interactions of these five requirements.

R4 (Move moves) is subsumed by R5 (MoveRight moves right). Under R5, *MoveRight* does always move the cursor, because of the condition $\mathit{nearest} \in \mathit{menu} \setminus \{\mathit{cursor}\}$. (And the other direction operations also move it, by symmetry.)

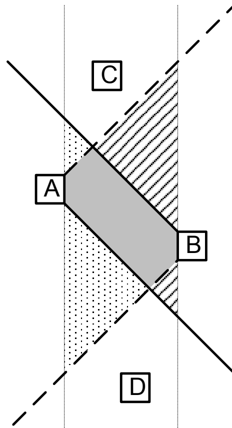

Fig. 4. Undo-consistent right-cone and left-cone, using the difference in x-coordinate metric.

So, under this particular R5, we can ignore R4, but if we modify R5, we must remember to revisit R4.

R3 (Undo) in combination with R1 (Deterministic) implies that each direction operation is not just functional, but *injective*. It defines *MoveLeft* as the compositional inverse of the deterministic (functional) *MoveRight*, so implies *MoveRight* is in fact an injection. Undo injectivity requires that each menu be the closest left item of its closest right item. That is, under the current formulation of R5, if A has B as its nearest item in its 'right-cone', then B must have A as its nearest item in its 'left-cone'. In terms of figure 4, B is right-closest to A because there are no other items in the grey and hatched shaded areas. For A to be left-closest to B, there must also be no menu items in the dotted area of B's left-cone. This is a very strong constraint, and clearly not all screen layouts need meet it.

R2 (Reachable) puts some further strong constraints on *Move*. It is not clear at this point if this is compatible with the other requirements.

4.2 Implementing the requirements

Ideally, the next step would be to calculate the weakest *MoveRight* predicate that satisifies all the requirements.

However, it is easy to see that this is not possible: consider a screen with two menu items in the first column, and one in the second (see figure 5). Whichever item the cursor is on in the first column, *MoveRight* will move it to the single unique item in the second; which clearly cannot be undone in an injective deterministic manner.

What to do? There are two possibilities: (a) strengthen *Screen* by putting more constraints on *menu* positions such that requirements do hold; (b) weaken (or change) some requirements conjectures so that they are implementable.

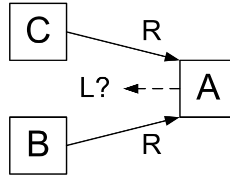

Fig. 5. Impossibility of injective undo under the 'right-cones' requirement

4.3 Constraining the menu layout

We discussed above how the undo requirement puts constraints on the menu layout (figure 4). If we were to enforce this constraint, would it be sufficient to satisfy all the requirements?

It is certainly not easy to see if such a constraint would guarantee reachability. Yet further constraints might be necessary. Before we expend effort in establishing such constraints, we should instead ask: is it reasonable to enforce this constraint? The answer is no, for two reasons. Firstly, it is very complicated: screen designers would not easily understand where they could place menu items in order to conform (particularly with Euclidean or Manhattan metrics, where the analogue of figure 4 is more complicated). Secondly, we actually cannot enforce it: screen designers can design whatever layouts they please, and we can make only recommendations.

So let us instead consider changing the requirements, to get an alternative 'intuitive' definition, but one that can be implemented.

4.4 Weakening the requirements

Which requirement to weaken? R1 (Determinisim) and R2 (reachability) are non-negotiable. R3 (undo) is very strong – forcing injectivity on each *Move* function component, yet it does seem very desirable. Let us consider R5 *MoveRight*: it is certainly the most complicated requirement to specify and understand, so it seems an ideal candidate for weakening. (We must remember to check that R4 is still subsumed by the new R5′.)

The existing requirements can actually help us formulate R5′. Consider plotting the path of a rightward moving cursor. Determinism means this path never splits; injectivity means no two paths ever join. So we can consider the *chain* of menu items described by this path. We want this chain to 'move right' across the screen, which we can specify by some predicate on the x components of the menu items. What happens at the right-hand edge? The cursor wraps around to a left-edge menu item. If we were to consider the entire chain of rightward moves, including several possible wrappings, we would find it difficult to specify the desired property. But if we consider the rightward chains and the wrappings separately, we can get an elegant specification of the desired property. Moreover, we can specify different kinds of wrapping for different circumstances. Figure 6 shows one possible set of 'rightward' chains. Such chains do also allow purely

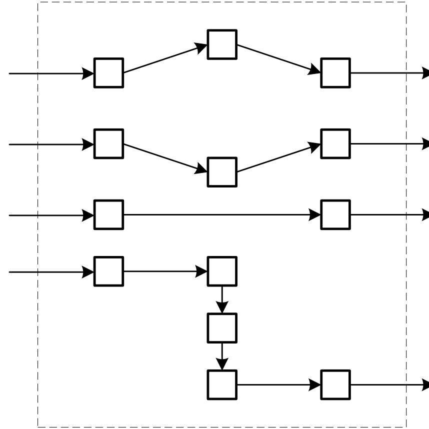

Fig. 6. a set of rightward chains covering all menu items on the screen

downward movements (which were not allowed by the earlier 'rightward cone' formulation of R5), to allow for sensible behaviour with vertically stacked menu layouts.

An *LRChain* is an injective sequence of *position*s (injective, so all the positions are different), of length at least two (to remove trivial chains, and to ensure R4). In the chain the x coordinates of the positions are increasing, or are the same with the y coordinates increasing.

$$\begin{aligned} LRChain == \{\ & s : \mathrm{iseq}\, position \mid \\ & 1 < \# s \\ & \land (\ \forall m, n : \mathrm{dom}\, s \mid m < n \bullet \\ & \qquad (s\ m).1 < (s\ n).1 \\ & \qquad \lor (s\ m).1 = (s\ n).1 \land (s\ m).2 < (s\ n).2\ )\ \} \end{aligned}$$

We augment the definition of *Screen* with a sequence of *LRChains*.

$$\begin{array}{|l} \textit{LRChains} \\ \hline Screen \\ lrchain : \mathrm{seq}\, LRChain \\ cl, cn : \mathbb{N} \\ \hline \{\ l : \mathrm{dom}\, lrchain \bullet l \mapsto \mathrm{ran}(lrchain\ l)\ \}\ \mathrm{partition}\, menu \\ \neg\ ChainsCross[lrchain / chain] \\ cl \in \mathrm{dom}\, lrchain \\ cn \in \mathrm{dom}(lrchain\ cl) \\ cursor = lrchain\ cl\ cn \end{array}$$

We require the positions in the set of chains to partition the *menu*: each menu position occurs in precisely one chain. The chains run from left to right across the

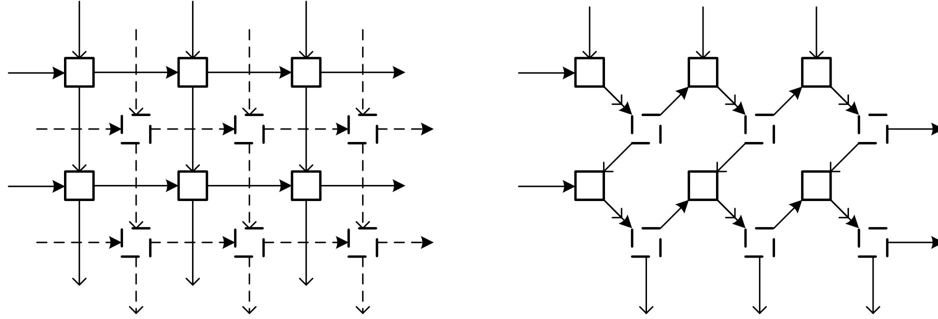

Fig. 7. two possible patterns of right chains and down chains (a) mutually unreachable sub-regions under R5′a; (b) full reachability under R5′a

screen by definition. We also require that the chains do not cross (see appendix for details). Finally, we define the position in the chains of the cursor.

There are (at least) two possibilities on wrapping: either the cursor moves back to the beginning of the same chain (a behaviour seen in simple DVD menues) or it moves to the beginning of the next chain (a kind of 'carriage-return/line-feed' behaviour suitable for text-based web pages). We capture each of these requirements in a separate conjecture.

R5′a: pressing the right arrow key causes the cursor to move to the next item in its chain, unless it is at the rightmost end, in which case it wraps to the beginning of the same chain. (See the appendix for the definition of the ++ operator.)

$$\begin{array}{l} \mathit{MoveRight};\ \mathit{LRChains} \vdash ? \\ \text{let } cn' == cn ++ \#(\mathit{lrchain}\ cl) \bullet \mathit{cursor}' = \mathit{lrchain}\ cl\ cn' \end{array}$$

R5′b: pressing the right arrow key causes the cursor to move to the next item in its chain, unless it is at the rightmost end, in which case it wraps around to the beginning of the next chain; if it is at the end of the last chain, it wraps to the beginning of the first chain.

$$\begin{array}{l} \mathit{MoveRight};\ \mathit{LRChains} \vdash ? \\ \text{let } cn' == cn ++ \#(\mathit{lrchain}\ cl) \bullet \\ \qquad \text{let } cl' == \text{if } cn' = 1 \text{ then } cl ++ \#\mathit{lrchain} \text{ else } cl \bullet \\ \qquad\qquad \mathit{cursor}' = \mathit{lrchain}\ cl'\ cn' \end{array}$$

The new R5′ does seem to capture the move right requirement adequately. In addition R5′b satisfies R2 (reachable), as it directly scrolls through all menu items in the order specified by the chains. However, R5′a does not necessarily satisfy R2: see figure 7a for a counter-example. The chains may have to be chosen carefully to ensure reachability (figure 7b).

So the diamond menu of figure 1 could be supplied with navigation chains as shown in figure 8.

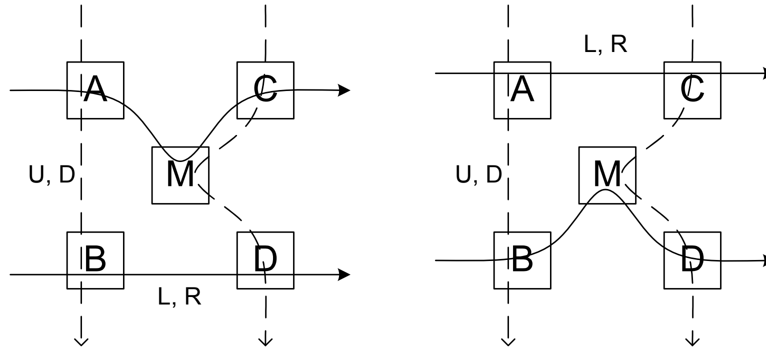

Fig. 8. two possible patterns of right chains and down chains for the diamond menu of figure 1

5 Conclusions and further work

The implementation of the requirements is now reduced to developing an algorithm that can find suitable chains that obey the chain constraints, and provide reachability. (Recall, this is the responsibility of the screen navigation provider, not of the screen designer, because there is no standard facility in the screen design mark-up to include navigation information.) Developing such an algorithm is beyond the scope of this paper, but clearly requires further refinements of the requirements to capture 'good' chains.

However, the original problem has certainly been reduced to a much simpler one. The chains approach automatically ensures R1 and R3–R5, and shows a simple way to detect if R2 is fulfilled (a simple 'mark and sweep' algorithm is sufficient to test reachability). What is next required is a proof that it is always possible to find some set of chains that fulfil R2, and then an algorithm to find them.

The 'requirements as conjectures' approach has allowed us to write a 'state and operations' style Z specification, to explore the properties of the operations without having to specify them in any detail, to capture requirements that cannot be expressed as properties of single operations (such as reachability), and to analyse the impact and consequences of particular stated requirements.

It might be argued that we did end up with the rather algorithmic-looking requirement R5′; however the existence of the required chains is merely asserted, and no algorithm for finding them is (yet) provided. The process of getting to this point allowed us to understand what the seemingly simple 'move right' property really entailed.

This paper shows the route to a solution for simple menu layouts. The full problem is more complicated. Menu items may be non-rectangular, complicating the intuition of which other item should be visited next (figure 1). Menu labels may affect the intuitive navigation order (for example, if they are numbers). However, we believe the approach here is applicable in more general cases.

6 Acknowledgements

We would like to thank Dr. Charles A. Whyte of ANT Ltd. for bringing the very real problem of DVD navigation menus to our attention, and the anonymous referees for helpful comments.

All the Z in this paper has been typechecked with Formaliser, and then undergone some minimal conversion to Standard Z syntax [ISO-Z 2002].

References

[Abowd & Dix 1992] Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.

[ISO-Z 2002] ISO/IEC 13568. *Information Technology – Z Formal Specification Notation – Syntax, Type System and Semantics: International Standard*, 2002. http://www.iso.org/iso/en/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip.

[Norman 2001] Donald A. Norman. DVD menu design: The failures of web design recreated yet again. December 2001. http://www.jnd.org/dn.mss/DVDmenus.html.

[Valentine *et al.* 2004] Samuel H. Valentine, Susan Stepney, and Ian Toyn. A Z patterns catalogue II: definitions and laws, v0.1. Technical Report YCS-2004-383, Department of Computer Science, University of York, October 2004.

[WC3] WC3. *HTML Techniques for Web Content Accessibility: Guidelines 1.0, WC3 Note 6 November 2000.* http://www.w3.org/TR/2000/NOTE-WCAG10-HTML-TECHS-20001106/.

A Utility functions and predicates

We use real numbers and associated functions (including the extension of addition and multiplication to the reals, real division, square roots, and so on), to capture various vector operations on screen positions interpreted as vectors. We use the definitions in [Valentine *et al.* 2004].

A *Point* is a pair of reals, representing a point in the infinite plane, or equivalently, as a vector from the origin to that point in the plane.

$$Point == \mathbb{R} \times \mathbb{R}$$

We define $(_ +_v _)$ as vector addition, $(_ -_v _)$ as vector subtraction, $(_ \cdot_v _)$ as vector dot product, and $(_ *_v _)$ as scalar multiplication on *Point*s in the obvious way (but omit the definitions here, for brevity).

$$\begin{array}{|l}
_ +_v _, _ -_v _ : Point \times Point \rightarrow Point \\
_ \cdot_v _ : Point \times Point \rightarrow \mathbb{R} \\
_ *_v _ : \mathbb{R} \times Point \rightarrow Point
\end{array}$$

The definition of the rightward viewing cone uses the cosine of 45 degrees:

$$cos45 == 1 \div \sqrt{2}$$

A unit vector derived from a general non-zero vector p is that unique vector that has unit length and points in the same direction as p.

$$unitVector == \lambda\, p : Point \mid \{p.1, p.2\} \neq \{0\} \bullet \\ (\ \mu\, u : Point;\ \alpha : \mathbb{R} \mid u ._v u = 1 \land 0 < \alpha \land \alpha *_v u = p \bullet u\ )$$

We define the distance between points. We leave this loose, because we may want to use a Euclidean metric, a Manhattan metric, the difference between the respective x coordinates, or some other measure. So we define the minimal constraints.

$$d_R : Point \times Point \rightarrow \mathbb{R}$$

$$\forall\, p, q : Point \bullet 0 \leq d_R(p, q)$$
$$\forall\, p, q : Point \bullet d_R(p, q) = 0 \Leftrightarrow p = q$$
$$\forall\, p, q : Point \bullet d_R(p, q) = d_R(q, p)$$
$$\forall\, p, q, r : Point \bullet d_R(p, r) \leq d_R(p, q) + d_R(q, r)$$
$$\forall\, p, q, r : Point;\ \alpha : \mathbb{R} \mid d_R(r, p) = d_R(r, q) \land 0 \leq \alpha \leq 1 \bullet \\ d_R(r, (1 - \alpha) *_v p +_v \alpha *_v q) \leq d_R(r, p)$$

The first four predicates capture the property of being a *metric*: it is positive, points are zero distance apart precisely when they are the same point, it is symmetric, and it satisfies the triangle inequality. The last predicate requires the metric to be *convex*: given two points p and q the same distance from a third point r, then any point on the straight line (defined in terms of the usual Euclidean metric on the plane) drawn between them is no further from r (it may well be closer).

Given two line segments, one defined by end points p and q, and the other by end points p' and q', the predicate *LineSegmentsCross* is true if they cross, that is, if they share a point in common.

LineSegmentsCross

$$p, p', q, q' : position$$

$$\exists\, \alpha, \alpha' : \mathbb{R} \mid 0 \leq \alpha \leq 1 \land 0 \leq \alpha' \leq 1 \bullet \\ \alpha *_v p +_v (1 - \alpha) *_v q = \alpha' *_v p' +_v (1 - \alpha') *_v q'$$

A collection of chains of line segments cross if any segment from one crosses a segment from another.

ChainsCross

$$chain : \mathrm{seq}\, \mathrm{iseq}\, position$$

$$\exists\, l, l' : \mathrm{dom}\, chain \mid l \neq l' \bullet \\ \exists\, m, n : \mathrm{dom}(chain\ l);\ m', n' : \mathrm{dom}(chain\ l') \bullet \\ \mathrm{let}\, p == chain\ l\ m;\ p' == chain\ l'\ m'; \\ q == chain\ l\ n;\ q' == chain\ l'\ n' \bullet \\ LineSegmentsCross$$

The infix operator ++ increments its first argument, unless it equals its second, in which case it returns 1. (It is similar to addition of 1 modulo n, except that it is based from 1 rather than from 0.)

$$\mathrm{function}\ 30\ \mathrm{leftassoc}(_ ++ _)$$

$$_ ++ _ : \mathbb{Z} \times \mathbb{N}_1 \to \mathbb{Z}$$

$$\forall\, i : \mathbb{Z};\ n : \mathbb{N}_1 \bullet i ++ n = \mathrm{if}\ i = n\ \mathrm{then}\ 1\ \mathrm{else}\ i + 1$$