




# Safer User Interfaces: A Case Study in Improving Number Entry

Harold Thimbleby

**Abstract**—Numbers are used in critical applications, including finance, healthcare, aviation, and of course in every aspect of computing. User interfaces for number entry in many devices (calculators, spreadsheets, infusion pumps, mobile phones, etc.) have bugs and design defects that induce unnecessary use errors that compromise their dependability. Focusing on Arabic key interfaces, which use digit keys  ,  usually augmented with correction keys, this paper introduces a method for formalising and managing design problems. Since number entry and devices such as calculators have been the subject of extensive user interface research since at least the 1980s, the diverse design defects uncovered imply that user evaluation methodologies are insufficient for critical applications. Likewise, formal methods are not being applied effectively. User interfaces are not trivial and more attention should be paid to their correct design and implementation. The paper includes many recommendations for designing safer number entry user interfaces.

“The most important property of a program is whether it accomplishes the intentions of its user.” Tony Hoare, 1969 [13]

**Index Terms**—Error processing, software/software engineering, user interfaces, human factors in software design, user interfaces, information interfaces and representation (HCI)

## 1 INTRODUCTION

PROGRAMMING is difficult. Over 50 years ago what are now called formal methods were developed so programs could be implemented that reliably achieved what their designers intended. Dijkstra memorably argued that debugging could only find bugs [6]: being unable to find bugs did not mean there were no bugs—no amount of debugging can prove the absence of bugs. One therefore needs to prove a program is correct. Hoare and others developed various rigorous formal techniques to reason correctly about programs without relying on debugging.

Dijkstra’s comment recalls Popper’s scientific philosophy of refutation [28]: Popper defined criteria for scientific theories and showed it is impossible to prove a theory correct by experiment. Both Dijkstra and Popper are right for the same reasons. It follows that in principle one cannot use empirical experiments to establish a user interface is correct.

Yet, according to the international standard ISO 9241, which defines best practice [14], user interfaces should be implemented, tested on users, bugs fixed, and then re-tested in an iterative, experimental cycle. This empirical process involves human participants and statistics to ensure sufficiently reliable conclusions are drawn despite natural variation in human behaviour and performance. Particular care has to be taken to ensure that the participants appropriately represent the final users of the system.

Building user interfaces to be used by people is a very different type of problem than building programs to be run

by computers. Designers cannot plausibly anticipate all user needs and requirements in detail, so a prototype is developed and tested on users. Indeed, users may change how they behave or change what they want after they start using a prototype, so the process has to be iterated.

Yet some user interfaces are safety critical and must be developed in ways that *must* avoid or mitigate safety issues. A balance needs to be struck: formal methods should be used to assure correctness and the absence of defects, and conventional usability experiments should be used to polish user interfaces and identify classes of defect that should then be proved absent. For example user experiments or expert heuristic analysis might identify the need for undo, then formal methods can be used to *ensure* that undo is available and works correctly in every state of the system.

Unfortunately, most user interface experts think formal methods are inaccessible and inapplicable, and most formal methods experts think user interfaces are trivial (which is an ironic consequence of “ease of use”). And a third group, many designers and programmers, just build user interfaces that are subject to neither usability nor formal scrutiny because they seem so simple they “obviously” work. User interfaces for number entry are a case in point.

The present paper is concerned with user interfaces for number entry, and specifically number entry using conventional Arabic numeric keys, as illustrated in Fig. 1, or as can be used with standard QWERTY keyboards. Such user interfaces are used for many purposes: dates and times, telephone numbers, passcodes for security systems, cash machines, finance and mathematics generally, as well as in numerous computer applications, from setting tab positions in word processors to scaling images.

Handheld calculators are a very familiar application of number entry, so they will be used to illustrate many design issues in this paper. By using real, clearly identified devices we demonstrate the techniques discussed scale to design

• The author is with the Department of Computer Science, Swansea University, Swansea SA2 0SF, Wales, United Kingdom.  
E-mail: harold@thimbleby.net.

Manuscript received 31 July 2014; revised 24 Nov. 2014; accepted 8 Dec. 2014. Date of publication 17 Dec. 2014; date of current version 17 July 2015.

Recommended for acceptance by H. Sharp.

For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.

Digital Object Identifier no. 10.1109/TSE.2014.2383396

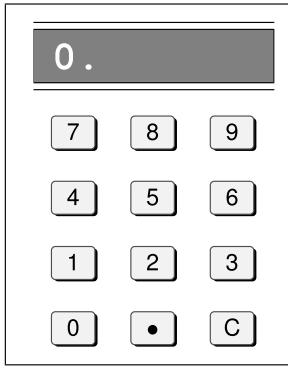


Fig. 1. Sketch of a simple numeric user interface of the type explored in this paper. Many alternative keyboard layouts are shown in Fig. 2. Note that a layout does not specify any *interaction* design decisions. For example, it is not possible to tell what the key [C] does; after pressing it, will 0. or 0 or nothing be displayed? Perhaps the [C] means “correct” rather than “cancel” and the display will change to an earlier display, say 123 or 12....?

issues that arise in real systems; the discussion is not limited to idealised systems. Furthermore, as consumer products, calculators are very easy to obtain to replicate and explore the defects examined in this paper. A note at the end of this paper briefly summarises all devices mentioned in this paper.

### 1.1 Contributions of This Paper

We use the term *rule* to mean a design property that can be used to express interaction properties precisely and that can be reasoned with. If a user interface is correctly implemented, then it will obey its rules, which in turn were derived from design requirements, themselves established by empirical experiments (or based in the relevant literature or experience of the domain of application). Crucially, designers (or the design tools they use) should be able to think clearly about rules, for example considering whether they are consistent and cover all possible cases of interaction. Rules cannot be seen by users, thus designers have an obligation to carefully select and implement appropriate rules.

We will show that rules user interfaces could obey can be stated using a simple notation, which we will introduce. One can then reason how to design safer and more consistent number entry user interfaces. Our notation is based on and is equivalent to Hoare triples [13]. It would have been possible to express the same issues in many other notations, such as HOL, PVS, SMV, TLA and VDM. However, many formal methods have steep learning curves, and there is a tendency to promote one over another because once you know a notation, using it seems much easier than learning an alternative. In contrast, our lightweight notation takes little effort to learn and can be used immediately. The disadvantage is that there is no tool support; there is no automatic way to ensure coverage, type correctness or other properties. Nevertheless, it is trivial to translate the notation into a tool-supported notation or programming language (like SPARK, which has assertions). The real contribution, then, is not so much the notation, but demonstrating that user interfaces are not designed rigorously, and that they could be and should be.

## 2 MOTIVATING EXAMPLES

We start with some broad-ranging motivating examples, which illustrate common design defects. Then, in contrast,

Section 3, shows that analogous problems were recognised over 50 years ago in programming, and for which there are now many ways to manage them or avoid them. Putting user interface design and programming one-after-the-other highlights that user interface design has not adopted the established benefits of formal methods.

### 2.1 Problems of Unclear, Unstated Design Requirements

Fu [7] points out a surprising lack of regard for the specification of requirements in medical device software, even though the field is safety critical and regulated. Devices that are not safety critical in a regulatory sense (such as hand-held calculators), even though they may be used in medical and other safety critical applications, fare even worse.





User interfaces implementing unstated, incomplete or inconsistent requirements will have defects, and probably unnecessary and confusing variation. Number entry has interesting problems: it appears to be simple, so designers may not bother to specify and analyse it adequately.

Numerical issues such as overflow interact with concrete display representations, such as field widths. Many number interfaces ignore excess input after the “end” of a number and some ignore “incorrect” keystrokes—if the user interface expects integers, 1.5 may be misread as either 1 or 15 depending on the implementation. Numbers may be syntactically invalid or out of range, but most user interfaces ignore errors (e.g., two decimal points) and happily process some valid number (e.g., the prefix up to the second decimal point, ignoring it and anything beyond it). No number entered may be converted to a default value (typically zero or a previous value) without the user being aware. And so on.

Except when the display is full, number entry displays behave as if digits are appended to whatever is displayed. This behaviour can be implemented in many ways: primarily, either as a string operation or as a numerical operation. As string concatenation, the meaning of the decimal point is that it is just a character. Alternatively, as a numerical operation, the decimal point is typically implemented as a flag (or a transition in control flow) that changes the meaning of subsequent digits. Amongst other differences, the behaviour of *two* decimal points will be different in the two methods of implementation. As a user will only rarely enter two decimals, these differences will be unfamiliar and possibly a surprise. Unpredictability is arguably one of the last things a user wants after an error.

Many users spend most of their time in general purpose environments, such as word processors and web browsers. In these environments, all input is simple text, so decimal points are treated no differently to digits, and the delete key deletes the previous keystroke. The user’s model that is acquired and reinforced in this environment does not work on number-based user interfaces: the keys [.] (decimal point), [+/-] (change sign) and [←] (delete) all behave differently.







Even simple-looking requirements for number entry may be inconsistent. The Institute of Safe Medication Practices (ISMP) has rules to improve the legibility of numbers [17]: “naked decimal points” are forbidden (e.g., because .5 may be mistaken as 5), and trailing zeros after a decimal point are forbidden (e.g., because 5.0 may be mistaken as 50). Unfortunately these well-meaning requirements cannot be








invariant: as a number is entered in a user interface it may go through error-prone intermediate stages. If a user enters    and pauses while entering it (for how long?), what should be displayed? If the ISMP rules are rigorously followed, the intermediate number should be displayed as  , but this makes the behaviour of the next keystroke, whether a digit or a delete key, ambiguous—which defeats the point of the ISMP requirements! One solution is for the display to flash or change colour so that invalid syntax can be visible [43]. In other words, despite their goals, unmodified ISMP requirements cannot be considered user interaction requirements: they are problematic in any number display that can be interacted with.


## 2.2 Problems of Logs

Many devices record a log of what they do, and this may be used to help understand incidents involving use of the device. For example, in a hospital if a patient receives an overdose of a drug, then the log of the infusion pump (a device that delivers drugs automatically) may be consulted to see if the pump delivered the overdose. This would then be evidence that the user instructed the device to do so.

Unfortunately if the infusion pump has a user interface design (as many do) like the Casio HR-150TEC calculator the following scenario is plausible:<sup>1</sup>

*User keyed*     —that is, the user accidentally keyed two decimal points and pressed  to delete the second decimal point.

*User thought*      =  , because  would delete the preceding keystroke, as it does on any PC application.


*Device logged* 5. The calculator records that the user entered the number  on its log.

If the user keyed something like this, continuing with the calculation would update the display and these keystrokes would be lost; the final result would be incorrect, and the log would show the user made an uncorrected mistake. The point of using a calculator is that you do not know what the answer is, and therefore few real users would be able to tell the difference between a calculation based on 0.5 and one unexpectedly based on 5. Many devices are similar: a number the user enters (e.g., a drug dose) is generally part of a larger interaction sequence, and in general it is very hard to spot an intermediate error.

One can imagine an incident investigator confronting a nurse with the log: “You told the infusion pump to deliver 5 mL of the drug, which killed the patient.” The nurse might say, “I thought I’d entered 0.5, but if the log says 5, I suppose I must have made a mistake.” Thus the nurse incriminates themselves. In fact, the device may have implemented delete like the HR-150TEC and, if so, its behaviour would have induced the fatal error and misdirected blame on the user. *Until user interfaces are implemented correctly, their logs cannot be believed.*

1. As explained in the introduction, we use concrete examples from specific devices, briefly summarised at the end of the paper, so that they may be easily replicated by the reader. Casio is a leading manufacturer and its devices are widely available and much easier to obtain than infusion pumps. All problems discussed arise on a wide variety of devices and are not restricted to any one manufacturer.

In fact, the HR-150TEC displays a decimal point all the time, regardless of whether the user has keyed one. It is likely, then, that the program code implementing the number entry user interface does not represent decimal points explicitly, and therefore it was problematic to implement the delete key as a general delete key. Rather, it is easier to implement it as an operation on a numeric value, ignoring the decimal point. This is exactly how delete behaves on the Casio.

Interestingly, the *entire* explanation of the delete key in the HR-150TEC user manual is the single concrete example “7 8  → 7 8” just correcting a single digit, and from which a user would certainly be justified generalising its behaviour to deleting other keystrokes. Perhaps the detailed behaviour of the delete key was overlooked?

## 2.3 Problems of Design Variation

There is considerable variability in user interface design for managing error: almost all user interfaces handle correct numbers correctly, but they vary widely on how they handle error, as illustrated in the examples above. Such arbitrary approaches to handle error will induce *transfer errors*: that is, over time, users acquire low level skills to correct error: doing such-and-such corrects an error and the user can continue. These strategies become automated and drop out of conscious attention. Hence on a system that behaves in a different way, in particular in any way that does not draw the user’s attention to the differences, the user is likely to automatically correct an error and make the situation worse. From the perspective of the present paper, it appears that the lack of consistency follows from failing to think through error handling; for example, we can imagine simple program code that implements “read a number” and, say, simply terminates when it parses an unexpected character as if it was the end of the number. In Section 3, below, we consider a classic programming problem, much simpler than reading a number, but nevertheless an error-prone example that illustrates the need for clearer thinking.

The Casio *fx-85GT* implements the delete key so it deletes both digits and decimal points; in the example above (Section 2.2), the user would have entered 0.5, not 5, after correcting multiple decimal points. This variation in user interface design will induce transfer errors. A user familiar with one Casio calculator will be induced to have problems with another. Unnecessary design variation *for the same task* seems to be confirmation that delete key behaviour has been overlooked.

Variation also occurs between device manufacturers. Below, three devices are compared handling the same sequence of keystrokes:






Key press	Casio <i>fx-85GT</i>	Casio HR-150TEC	Apple iPhone
		0.	0
	0	0.	0
	0.	0.	0.
	0..	0.	0.
	0.	0.	0
	0.5	5.	5



TABLE 1  
There Is No Common Way of Handling Negative Numbers (Section 2.5)

Key sequence	Canon F-502G	HP EasyCalc 100	HP SmartCalc 300s	Apple iPhone	Apple OSX	Casio OfficeCalc 100	Samsung Android
$\pm$ 9	9	9	-9	-9	9	9	-9
6 7 $\pm$ $\leftarrow$	-6	-6	67	-6	-6	-6	-6
$\pm$ 0 $\leftarrow$	0	0	any number	-NaN	0	0	Wrong format
9 $\pm$ $\leftarrow$ $\pm$	0	-0	Syntax ERROR	Error	NaN	0	
9 $\pm$ $\leftarrow$ 8	8	8	98	-NaN	-8	8	-8

"Any number" arises because the SmartCalc inserts `Ans`, a variable denoting the previous answer, making the input syntactically correct but numerically arbitrary. "NaN" means "not a number," a bug that should have been detected by the device and reported (e.g., as `Error` to the user) or should have been avoided altogether.

## 2.4 Problems of Ambiguous Display Feedback

Part of the problem with decimal points is that the display does not unambiguously show the user how many decimal points have been keyed. Many number user interfaces display `0.` when they are switched on or cleared, and the display does not change when the user keys `0` or `.`. Unfortunately, this is ambiguous: if the user keys `5` next, the display may change to `0.5.` or to `5.`.

The *fx-85GT* has a left-justified display, which ensures that deleting a key *always* removes the right-most character from the display, whereas on the more common right-justified displays deleting a key moves the entire display contents right. Deletion when the display shows `55.` cannot provide unambiguous feedback to show whether the 50 digit or the 05 digit was deleted.

Some calculators, including the HR-150TEC, use  $\blacktriangleright$  as their representation of the delete key, which makes sense as pressing  $\blacktriangleright$  moves the display contents to the right as it deletes the right-most digit (provided the display is not showing just `0.`). However, on the HR-150TEC, the key  $\blacktriangleright$  will move the display right even when it is an answer to a calculation, in which case  $\blacktriangleright$  is not deleting what the user keyed!

## 2.5 Problems of Negative Numbers

Not all number entry user interfaces support negative values, but for those that do there is a potential conflict with conventional mathematical notation. On many calculators starting a new expression with an operator like `+` adds the next value to the previous result, which implies that starting a calculation with `-` is ambiguous: it could mean start a negative number or subtract a number from the previous result. Most calculators resolve the problem by providing an unconventional key for negating numbers, like  $\pm$  or `(-)`.

Every calculator examined allows  $\pm$  to be used anywhere within a number. Thus  $\pm$  5 0, 5  $\pm$  0 and 5 0  $\pm$  are equivalent ways to enter `-50`. However, there are major variations with how  $\pm$  interacts with delete (see Table 1), e.g.,

- On the Hewlett Packard EasyCalc 100 calculator, the delete key ignores the  $\pm$  key and deletes any preceding digit. Hence `6 7  $\pm$   $\leftarrow$`  is `-6`, not `67`.
- On the Apple iPhone, `AC  $\pm$  0  $\leftarrow$`  displays `-NaN` ("NaN" stands for "not a number," and being visible to the user indicates a bug).
- On Apple OSX, the calculator does not allow this: it will not display `-0`; entering  $\pm$  9 results in `9`,

so a prefix  $\pm$  is ignored; yet `9  $\pm$   $\leftarrow$  8` results in `-8`, even though immediately after the delete the display is an unsigned `0`. The internal negative flag is incorrectly programmed.

- `C 9  $\pm$   $\leftarrow$   $\pm$`  results in `NaN` on OSX but `Error` on the iPhone. Two pieces of code with the "same" functionality from the same manufacturer exhibit different bugs.
- When we tried to understand the behaviour of `-` on the HR-150TEC, it froze until it was switched off and on again.
- On the HR-150TEC, the sequence `4 - 5 =` results in `1` (when perhaps `-1` was expected), because the `-` keystroke turns the `4` to `-4`, then the `5` is the next number added to it. This unusual behaviour ensures that learning one calculator will be of little help for using another.

It is interesting that two Apple calculators work in different ways for something so "simple," but this is not unusual—elsewhere in this paper different models from the same manufacturer have incompatible user interfaces, and Fig. 2 shows the "same" model from a single manufacturer may be available in several incompatible variations.

The mixture of incompatible, confusing user interface design and actual bugs suggests that manufacturers have not attempted to specify the requirements for negative numbers, made careful design trade-offs, nor attempted to implement them carefully or correctly.

## 2.6 Problems of Input Field Overflow

Grete Fossbakk made a typing slip and accidentally transferred \$100,000 of her money to an unknown person who spent it [26]. With an accidentally repeated 5, she typed 12 digits into an account field, but unfortunately the first 11 digits of the number was a valid Norwegian account number, even though the full 12 digit number itself was an invalid account number.

The repeated 5 may have been caused by a faulty keyboard or software rather than a keying slip, though presumably she was using her own PC and did not use the bank's PC, so the keybounce may have been technically her responsibility (if one agrees with the various waivers manufacturers impose on users). In other areas, key bounce is recognised by regulators as a regular and serious problem, which has resulted in product recalls and seizures [15].

In user studies to explore how Fossbakk made the error [26], 41 percent of numbers entered were too long. It is surprising that the bank does not check for such a common

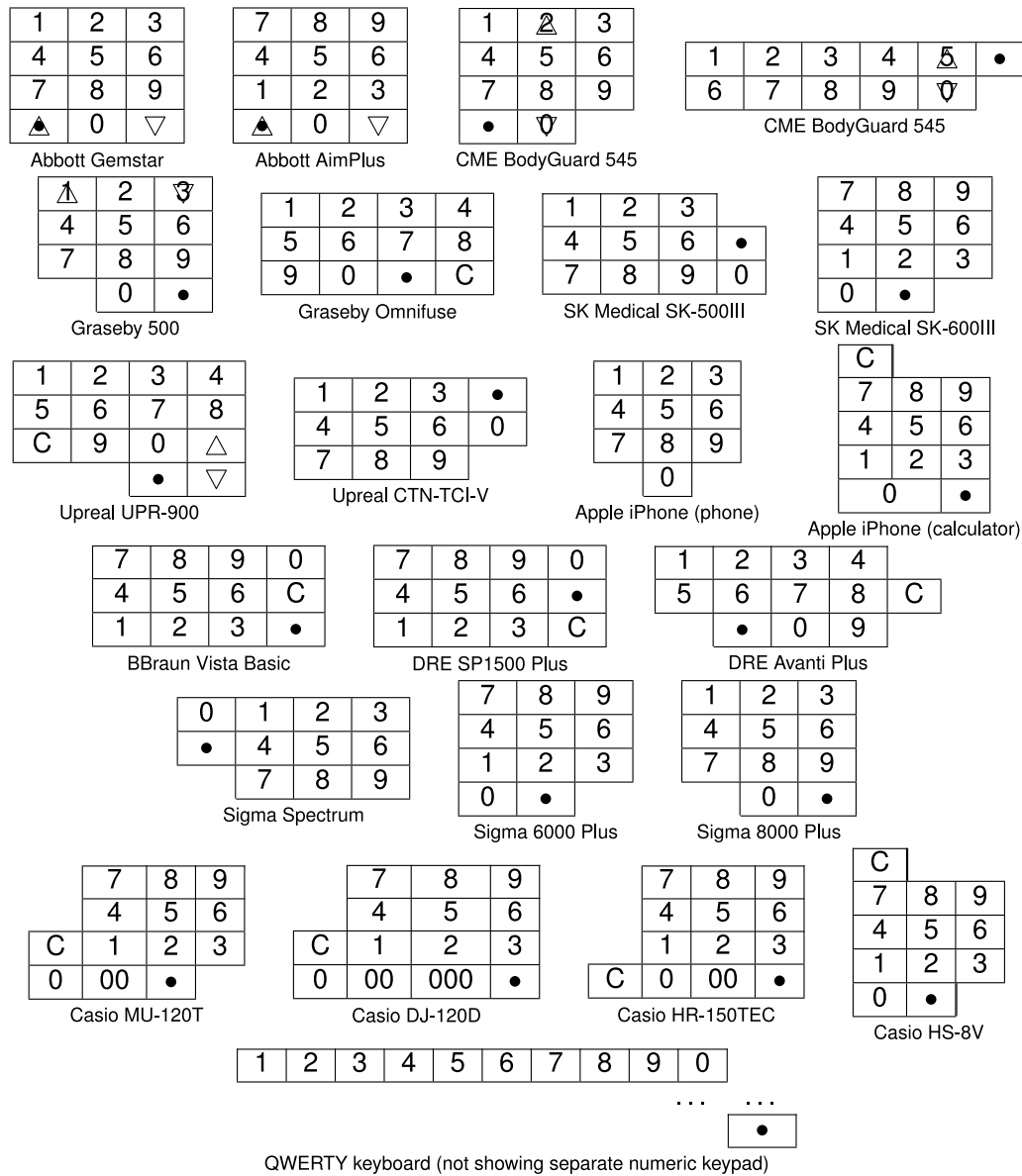


Fig. 2. Keyboard layout on a selection of number entry systems, showing that the same manufacturer and even the same model use different layouts (e.g., see the two variants of the CME BodyGuard 545, top right). The variety of keyboard designs will cause confusion in use, training and maintenance. The schematics make the decimal point much clearer than on most devices; the schematics do not show the variety of uses of the “spare” button locations, which are variously blank, “info,” etc. The  $\Delta$  and  $\nabla$  represent up/down keys, in many cases dual uses for numeric keys, which is known to cause confusion [16]. Note that telephones generally have a top line of 123 in contrast to calculators, which generally have a top line 789; on mobile phones and wristwatches (which can run calculators as applications) this is likely to cause unnecessary confusion.

error. Amusingly, in its defence the bank argued that there should not be different rules of responsibility depending on the length of a number!

In 2013, my own Lloyds Bank internet account uses an HTML text field for account numbers, defined as:

```
<input type="text" autocomplete="off" ...
  maxlength="8"
/>
```

The parameter `maxlength="8"` ensures the browser will discard any characters typed in excess of 8. Again, the logs cannot show an auditor whether the user typed an invalid account number that should have been rejected.

Number overflow can occur for many reasons. We might be interested what proportion of the world’s population is Welsh (I live in Wales). The population of Wales divided by the population of the world can be

found using a calculator:  $3,063,500 \div 7,300,000,000$ . The following results are obtained:

Casio HS-8V	0.04 ...
Apple iPhone portrait	0.004 ...
Apple iPhone landscape	0.0004 ...
Hewlett Packard EasyCalc 100	0.0004 ...

All ignore digits that do not fit into the display—and none report an error when they discard digits. The display is eight digits on the Casio HS-8V, but the world is 10 digits, so the division is out by a factor of 100; the iPhone in portrait has a display with nine digits, so the answer is out by a factor of 10; and the iPhone landscape display is larger than 10 digits, so it gets the answer right. The EasyCalc would have similar problems with calculations involving more than 12 digits, but the calculation here does not reveal it.

Whatever is going on inside the Apple iPhone, arguably it could have detected an error since it provides two different answers. It could report an error when there is a discrepancy. Probably the iPhone has a single “calculator engine” and in portrait mode the user interface fails to tell the engine what the user keyed after the first nine digits. In this case, the engine cannot do any better, as it is being let down by the user interface, which in turn is letting the user down—it is discarding input without any warning.

## 2.7 Problems of Behaviour Depending on Value

The Baxter Colleague 3 volumetric infusion pump has a numeric user interface: if the user enters  then the display shows **10.1**; if the user enters  then the display shows **1001**, with no warning or error sound that the decimal point has disappeared. In other words, when the number entered is “large” the user interface silently ignores decimal points—the number entered will be a factor of 10, 100, etc., higher than the intended. This is a design defect found on many medical devices [22].

Instead of using numerical values, the Sigma Spectrum uses a character count limit, so it might accept  but if  is keyed, it will not accept further digits; thus forcing the number displayed to end in a decimal point (forbidden under Institute of Safe Medication Practices rules [17]). Despite numeric rounding being well known (and appropriate in the domain) it will treat 100.9 as 100 not 101.

## 2.8 Problems of Changing or Editing Values

Often a user will want to change a number; typically the device will display the last number accepted in the same place the user will edit or enter the new number. The Alaris PC illustrates one problem: if the display shows **9** from previous use, then pressing  will change the display to **0.1**, but if the **9** is a number currently being entered, then pressing  will change it to **90.1**. Confusion arises because the Alaris PC does not distinguish the previous and current numbers.

This is a failure of equal opportunity [30], which says input and output should be exactly the same; here they look the same but behave differently, which is a recipe for confusion.

If the user enters a number that is out of range the device will not accept it, yet the number is displayed in the same place as the previous number, which was in range. Thus the Alaris PC discards the final digit the user keys; if the user tries to enter 88888 when 9999 is the maximum, the display will show 8888, dropping the last keystroke. Underflow presents similar problems: the display might show **0.**, warn that 0. is less than the minimum (perhaps set at 0.1), yet forbid the user keying  which would make the number larger than the minimum—the problem is that once underflow is detected, the number displayed has been “accepted” and is no longer the number the user is entering (yet the display is identical)!

The Alaris PC requires fractional numbers to be entered starting with a decimal point; a leading zero is therefore an error. Hence if the display is **123** and the user keys , it is ignored except for a beep. Yet

continuing and pressing  changes the display to **0.** as if the discarded zero was in fact processed. (This design flouts the Institute of Safe Medication Practices rules; see Section 2.1.)

The user interface would be simpler and more consistent if the old values were never displayed when a user is entering (or about to enter) a new value, and if the number the user is keying is always faithfully displayed, regardless of overflow—then the standard correction keys will work as the user expects. As implemented on the Alaris PC, the device provides *some* correction, but thus making its behaviour unpredictable.

## 2.9 Problems of Unusual Behaviour on Errors

Many numeric user interfaces (such as the HS-8V) ignore excess decimal points, so  is treated as **1.23** without any error warning. The Graseby 3400 is unusual: it treats decimal points as clearing the decimal part of a number, so  is treated as **1.3** [38]. In Excel,  is treated as zero without warning if it occurs in a SUM expression. Many PC applications end a number at the first non-numeric character, thus ignoring the error—JavaScript, which underlies most web applications, treats 1..2 as 1, because an unexpected decimal point ends a number without warning. Many examples are provided in [40], which more generally illustrates the large problem of system design that is heedless to error of all sorts.

## 2.10 Problems of Confusing Key Clicks

Many devices (e.g., the Alaris PC) provide different sounds when keys do different things; in particular an attentive user can tell by the different sounds whether a key press is being ignored. The Sigma Spectrum provides a “key click” sound when any key is pressed *whether or not* the key does anything; so (for example) keying 0.15 sounds exactly the same as if all keys were handled correctly, but the device only shows **0.1**.

## 2.11 Problems of Terminating Number Entry

Typically there is a key  to confirm a number has been completed; when this is pressed, the device records the number and goes on to its next activity. Often any non-numeric key confirms the number, but on some devices that allow many numbers to be entered, such as the Sigma Spectrum, pressing  changes the selected number—except that it does not “confirm” any number. Instead, the number reverts to the previous value before the user started entering it. In other words, while a user might think  is just a “passive” cursor movement (i.e., with no side-effects) it behaves instead like .

## 2.12 Problems of Time-Outs

Devices cannot tell whether a user has given up interacting with them. Battery-powered devices have battery life to conserve. Walk-up-and-use devices (like cash machines) do not want the next user to continue with the interaction started by the previous user. The solution is typically for the device to switch off or revert to standby after no use for some minutes.

The EasyCalc, despite having a photocell to provide power, switches off after 5 minutes of inactivity. It does not beep or otherwise warn the user it is about to switch off, and on switching on the displayed number is lost but, strangely, the memory register is not lost—so there is no technical reason not to save the displayed number too. At least the display goes blank so the user can see the device has reset.

In contrast, the Graseby 3400 [38] has a 4 second time-out that zeros the number currently being entered. Hence, entering  $\boxed{0} \boxed{\bullet} [\textit{delay}] \boxed{5}$  will enter  $\boxed{5}$  instead of  $\boxed{0.5}$ .<sup>2</sup>

If there is an argument for  $n$  second time-outs, then there is a pretty good argument for  $n + 1$  seconds—and so on! Since there is no perfect time-out interval, a better idea may be to flash and beep and try hard to recover the user's attention; if necessary there then might be a hard time-out, when the device has reason to give up hope.

### 2.13 Problems of Feature Interaction

Features seem useful, so combining features seems even more useful. Yet features may interact with each other detrimentally.

The HR-150TEC has a double-zero key  $\boxed{00}$  feature, to speed up entering numbers with repeated zeros. Unfortunately the key is handled specially by the delete key: pressing  $\boxed{00}$  then  $\boxed{\leftarrow}$  is treated as  $\boxed{0}$ . The reasoning is presumably that the  $\boxed{00}$  may be pressed when  $\boxed{0}$  was intended, so  $\boxed{\leftarrow}$  corrects *that* specific error rather than deleting the previous keystroke, which is its normal meaning. Do the new types of error and confusion offset the gains of the button? Unfortunately, having a key  $\boxed{00}$  might sell more calculators, and design trade-offs may then be secondary to sales.

### 2.14 Problems of Transient Error Warnings

Eye tracking experiments [25] show that the user does not pay perfect attention to the display. It is therefore advisable that error states are persistent and cannot be unset accidentally, and it may be advisable for errors to be associated with noises or vibration so that the user is made aware of them more effectively than just displaying a visual symbol.

The Apple iPhone calculator given the incorrect calculation  $1 \div 0. + 42 =$  will present the answer  $\boxed{42}$ —but it transiently displays **Error** when the key  $\boxed{+}$  is pressed. The user is likely to miss this warning, as they are concentrating on pressing the correct keys not on tracking the display. If the calculator had persistent error warnings, the warning would still be there when the user is ready to read the answer: the answer should be **Error** not  $\boxed{42}$ .

Many calculators display **E** (meaning “error”) on the far left and will display *any* number right-justified in the main part of the display. It is possible for a user to read the number and think it is the answer without noticing the **E** at the other end of the display. Therefore the main part of the display should either be blank or, preferably, display **Error** or equivalent warning. In some applications it may help the user further to display **Error! Press AC**, or otherwise clearly prompt the user that to proceed they

must clear the error. (The word “Error” can easily be written using seven segment displays, so it could be reprogrammed for existing systems.)

### 2.15 Problems of Inconsistent Ergonomics

The ergonomics, layout, and presentation of number entry user interfaces is clearly critical—poor lighting, poor tactile feel, poor font have all be criticised (e.g., [41]). There is classic research such as [5] which could inform design (or more research) so the diversity evident in Fig. 2 suggests that criteria other than usability and dependability drive user interface design for number entry—the diversity must increase transfer errors. Plausible design considerations in use include branding, compactness (i.e., weight, cost), business engineering (once a user is familiar with a particular user interface, any other user interface will seem hard and error-prone in comparison), confusion marketing, etc.

When a user presses a key, the device should provide feedback. Many devices have physical keys that feel they “click” when they are pressed, and many devices generate an audible click either from the mechanical movement or generated by software. Of course, successfully pressing a key is different from successfully achieving the intended action on the device. For example, it would be confusing if the display was full, but pressing  $\boxed{2}$  gave all the feedback as if it had been successfully entered. Devices should therefore provide more than “mechanical” feedback, and should make appropriate non-keyclick sounds when keys fail to work normally—if the display does not change when a key is pressed, there should be a warning.

### 2.16 Idiosyncratic Variations

Burglar alarms are “walk up and use” user interfaces, so the user might break off or start entering a number at any point, and they are therefore often permissive [37] in when a number starts. Since a typical alarm code is four digits the last four digits the user keys is taken to be the number entered. Thus there is no overflow; the most significant digit just disappears when the next digit is keyed. This style of “scrolling” interface is also surprisingly common in other contexts where it is clearly inconsistent with the rest of the user interface design, and where there is no walk up and use requirement.

Other variations are common too. Although the Apple OSX System Preferences allows Arabic number entry, it sets times in an idiosyncratic way. The display shows a valid time, such as  $\boxed{9:45}$ , and when the user selects it to enter a number, either the hours or minutes is selected. For example,  $\boxed{9:45}$  shows the minutes is selected. Pressing digits now *replace* the selected number; thus pressing  $\boxed{9}$  will change the display to  $\boxed{9:09}$ ; not to 59, and with the 4 silently lost. To enter a time like 12:45 they have to select hours, press  $\boxed{1} \boxed{2}$  then change the selection, then  $\boxed{4} \boxed{5}$ . Moreover, there is a time dependency: if  $\boxed{2}$  is entered, the time becomes  $\boxed{9:2}$  then a moment later,  $\boxed{9:02}$ ; if before a time-out, pressing  $\boxed{3}$  will change the display to  $\boxed{9:23}$ , but pressing it after the time-out, there is a beep and the display is unchanged. Within each component, digits move *leftwards*, suggesting one might start entering a time like 12:45 in the minutes component, but the colon does not work and just beeps. If the user enters 12:45 with a delay between the  $\boxed{4}$  and

2. A 4 second time-out seems very short for this user interface. It is plausible that internal hardware uses a 4 second time-out (e.g., to check that the motor has not stalled) and inappropriately, as a side-effect, the same mechanism resets the user interface if it has “stalled.”



5, the display becomes 12:04, then pressing the 5 will make it display 12:05.

Then there are erroneous examples: if the user tries entering a “time” such as 57:96 the display will be 07:09—the second digit of the first erroneous component is kept, but the first digit of the second component is kept. If the user enters 1,259, it may be displayed as 09:30, with an unchanged minutes setting. And so on.

We have not explained all of its features, and we are not sure we have understood what we have explained. A user has to read the display to check whether their intended number has been entered correctly. Similar problems occur in date setting user interfaces, with the added complication that day, month and year numbers mutually interact in a way that minutes and hours do not.

### 2.17 It Is Not Just Numeric Keypads ...

This paper focuses on numeric keypads, but there are many other forms of user interface for number entry. For example:

- The GE Dash 4000 uses a knob to adjust number values; turning the knob clockwise will increase values, anticlockwise will decrease values. So if a number displayed is 40, turning the knob clockwise will show successively 41, 42, 43, 44, 45, then 50, 55, 60. Turning the number back, anticlockwise, will show 55, 50, 45, 40—skipping values, not reversing the effects of the preceding clockwise rotation.
- The BBraun Infusomat [4] uses four keys to enter numbers: two allowing a cursor to be moved left and right, and two for digits to be increased or decreased. If the display shows 0. a user can move the cursor to the hundredths column, increase the digit by 1, yet 0.10, not 0.01, will be displayed—10 times out from what the user entered, but without warning.
- Using handwriting with immediate recognition feedback improves error rate [45].

## 3 PROGRAMMING MAXIMUM

The preceding section raised concerns with user interfaces, yet analogous concerns in programming are taken very seriously. In this section we present a familiar programming example to contrast the type of formal thinking routinely applied to program code to gain the sorts of detailed insights that are evidently lacking in user interface design. We take it for granted that we should reason formally about programs to ensure they are correct; we should not balk at reasoning about user interfaces.

We could have chosen reading a number as an example, but the code to do so would be distractingly long; instead, suppose we wish to write some code in Java to simply find the maximum value of an array `a` of integers. Here is how it might be written:

```
1.1 int max = 0;
1.2 for ( int i = 1; i < a.length; i++ )
1.3   if ( a[i] > max ) max = a[i];
```

*Testing is not sufficient.* Many tests of this code will show that it finds the maximum value correctly. It is possible that

“thorough” testing overlooks the critical cases that are incorrect. Indeed, there is a problem of circularity: if you write a program to find the maximum value of an array, how are you going to check it is doing the right thing, since checking is subject to the same blindspots that led to any errors in the program in the first place? One might resort to multi-version programming (i.e., using many “independent” teams of programmers), but this has been robustly criticised as a flawed approach [19].

*Formal reasoning is essential.* Thinking mathematically about a program is more reliable than testing. Here, it would reveal two flaws that testing may overlook. First, if the array consists of only negative numbers, the code cannot give a maximum value less than 0; it is therefore incorrect. Secondly, the value `a[0]` is ignored. Both problems can be corrected by replacing line 1.1 with `int max = a[0]`.

In fact, with line 1.1 as `int max = a[0]` the invariant `max = maximum(a[0..i])` is established, and each iteration of the for loop ends with `max = maximum(a[0..i])` established, and `i` increases in steps of 1 up to `a.length-1`, so on termination of the loop we have `max = maximum(a[0..a.length-1])`, which is what we want.

Formal reasoning would also beg to include the requirement “and the array `a` is unchanged,” as the faulty code

```
2.1 int max = 0;
2.2 for ( int i = 0; i < a.length; i++ )
2.3   a[i] = 0;
```

is otherwise a correct way to ensure that `max` is the maximum value of the array!

In summary, we have shown, as is well known, that programming is deceptively hard, and that formal reasoning increases the confidence that programs indeed implement what they are required to do and, concurrently, we also improve our understanding of what we want them to do. Conversely, without formal reasoning it is unlikely—literally, there is no reason—a program will do what is or should have been intended, although it might deceptively look like it does.

### 3.1 Lessons from Programming and Formal Methods

We conclude Section 3 with four insights:

- User testing is not thorough (it does not guarantee coverage) and it does not identify all possible problems of a design. Although user interface evaluation is, or should be routine, it focuses on what users can experience in a short time. This will help identify confusions and help improve user experience (UX), but it does not have the reach to identify all bugs that will eventually affect some users.
- Without rigorous reasoning it is very unlikely any user interface will do what it is intended to do. Bugs in user interfaces are hard to see and understand—and unlike program code, the behaviour of a user interface cannot be seen or read as a text. It has to be represented in other ways.
- Formal methods is not just reasoning rigorously about programs, but also about what we want them to do. In the maximum example, an “obvious”



invariant was not tight enough to specify what the intended requirement of the program really was.

- An intermediate approach, between user testing and formal methods, is to employ stochastic testing based on human error models. Here, simulated user trials explore large, complex state spaces. This has speed and coverage advantages over human evaluation, and is simpler than formal methods; its twin disadvantages are that a simulation cannot have the qualitative insights human users will have, and unlike formal methods, it cannot help design out errors as it can only help find them—and to find an error, one needs a preconceived concept of what the error might be. We do not discuss stochastic methods in the present paper, but see [2], [4], [39].

## 4 PREVIOUS WORK

For many years around the 1980s, calculators were a standard object for research in human-computer interaction (HCI); notable papers include [9], [23], [47]. The primary concern was usability and understanding the relation of the user's model to the device model. It is surprising that the problems reported in the present paper were not highlighted by this original 1980s research. In fact, HCI techniques seem insufficient to identify safety problems. Moreover, the devices that are studied in HCI are often devices that the experimenters are very familiar with, and therefore there is a possibility that both experimenters and experimental participants share the same blindspots.

Thimbleby and Cairns [3] showed that many user interfaces for numeric data entry ignore syntactic issues (for example, allowing numbers with more than one decimal point). They showed that restricting user input to syntactically valid numbers would reduce unnoticed errors. The numeric syntax of [3] was later generalised to arbitrary regular expressions [43]. Thimbleby [35], [36] reviewed problems with calculators specifically, and with Thimbleby [45], [46] proposed novel solutions that overcome many of the identified problems with calculators.

### 4.1 Formalising User Interfaces

Since the 1980s there have been attempts to formalise user interface requirements [11], but these have not become mainstream because the level of mathematical sophistication seems out of proportion to the potential gains in user interface design quality. More recently developments in automated reasoning tools, such as theorem provers, have meant that original user interface program code can be semi-automatically checked for user interface properties like “predictability” [21], [22]. There is an important conference series on formal methods and user interfaces, the ACM SIGCHI Symposium on Engineering Interactive Computing Systems [27] and its predecessor DSVIS [8]. While exciting that user interface properties can be formally verified, the skill required is still considerable and the resulting research remains opaque to many practitioners.

The present paper focuses on formal reasoning for number entry user interfaces. It identifies and solves

many problems that both the preceding HCI literature and the formal methods literature has missed. This is an interesting blindspot: if people cannot see problems, it does not matter whether formal methods or empirical methods are used—there is still a blindspot!

Our present interest in number entry user interfaces arose through very detailed examination of user interfaces for hospital infusion pumps, “simple” devices that deliver drugs automatically to patients after nurses have entered relevant dosages. After our first study [38], we have found that almost all infusion pumps have number entry problems, and the problem extends to almost all number entry interfaces of all sorts [3]. While our previous papers identified the problem and discussed its impact, we did not present a way to reason more reliably about user interfaces so the problems would not arise in the first place.

### 4.2 Human Error

Reason's *Human Error* [29] is a landmark book. He taxonomises human error: *violations* are actions that should not happen (for example, the user sets out to perform a criminal activity); then other forms of error can be broadly classified as intentional or unintentional. An *intentional error* occurs when the user mistakenly intends to do the wrong thing; in the context of the present paper, intending to set an infusion pump to 28.8 mL per hour and successfully doing so—when the correct rate should have been 1.2 mL per hour—is an intentional error.

*Mistakes* occur when the user correctly does the wrong thing, perhaps due to a misunderstanding or lack of knowledge. For example, a user might mistakenly believe that **DEL** deletes the last key they press. They would then be mistaken on many (but not all) devices—see Section 2.2 for examples.

In contrast, *slips* and *lapses* are unintentional errors that the user is unaware of. If a user performs the wrong operation, this is a slip (e.g., pressing the wrong key because their finger slipped); and if the user omits an action (e.g., by oversight) then this is a lapse.

These are human taxonomies; from the engineering perspective, the issue is whether the errors can be blocked or managed, and if so, whether the computer or other agent manages the error. For example, a violation is typically a security problem that can be mitigated by requiring passwords and keeping logs (if the latter, so that if a user chooses to perform a violation they know they will have to face recorded evidence). Accidental errors can be mitigated by practice, redundancy, safety checks, undo functions, and so on. An example of redundancy would be to require two users to enter a critical number, and to have a reconciliation process if the numbers do not agree. Another example would be setting an infusion pump not just to a rate, but also specifying the drug, the patient weight, the concentration and the intended duration, etc; if the infusion pump can work out that a drug is to be infused at an inappropriate rate for a patient it can block it.

An intentional error can follow an unintentional error and vice versa. For example, making a slip or lapse while using a calculator will result in the calculator showing the wrong result. This result may then be the number that is

subsequently used. The use of the incorrect number is then an intentional error.

Reason additionally defines *latent errors*, oversights in a design that “wait” for unanticipated conditions. Program bugs are obvious latent errors, but many are more subtle and lie in the requirements.

Many human errors occur predictably. For example, if we say we will see you at 7:00, that is all we need to say to you. But if we tell our alarm clock “the same thing” namely to ring at 7:00 it will take additional steps for it to register the instruction. The alarm clock cannot tell the difference between entering 7:00 and entering 7:00 *and finishing*. As it were, it might be “thinking” that we might still adjust the time so we have to explicitly confirm it, even though it is an unnecessary step in human-human interaction. Similarly, on most calculators, one cannot calculate  $4 + 5$  by just pressing  $\boxed{4} \boxed{+} \boxed{5}$ , as a final step  $\boxed{=}$  is required. There is evidence that “device oriented steps” are more error prone [1], but this research does not define device oriented steps (e.g., is pressing  $\boxed{=}$  a requirement of the task, or a device oriented requirement of a calculator?). Avoiding the step makes users more accurate [45].

In this paper, focusing specifically on number entry, violations and intentional errors are out of scope (they are typically handled either before or after a number has been entered); slips and lapses, on the other hand, occur frequently *during* number entry, and we need engineering techniques to help the user detect and manage them.

### 4.3 Error Correction

While human factors research focuses on the sources of error, what happens next is often more important. A user may make a slip for any reason, but if it is noticed and there is a way to correct it, the final outcome will be correct. Hence designers should focus on reducing adverse outcomes (e.g., patient harm) [42]. Viewed from this perspective, an important distinction is whether error is noticed or not and whether the user or the system first notices it, and if so, what can be done about it.

An error may lead to a bad outcome. How this may be quantified to inform design trade-offs depends on the domain. For example, an incorrect bank account number is either invalid or another account number (see Section 2.6) whereas if the number is a drug dose the relative error or a measure of the patient outcome (e.g., in quality adjusted life years, QALYs) is more insightful. Elsewhere we have compared user interfaces using expected relative error [3], [4], [25].

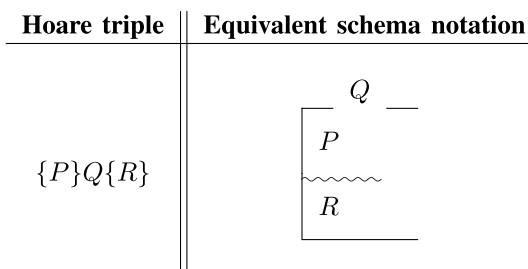
## 5 RULES FOR NUMBER ENTRY

A computer program executes a sequence of statements, much like a user executes a sequence of commands to control a user interface, typically by pressing keys or tapping a screen. The program code  $A; B; C$  behaves like the user “program”  $\boxed{A} \boxed{B} \boxed{C}$ . Hoare’s insight [13] was that a formalised process can be used to prove that if certain conditions  $P$  hold and a program  $Q$  is executed and

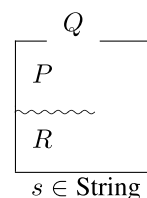
terminates, then certain conditions  $R$  will hold. The relation may be written  $\{P\}Q\{R\}$  in the modern Hoare triple notation. Depending on the application, we may wish to prove  $R$  holds, we may wish to derive a correct program  $Q$ , or we may want to weaken  $P$  in some way so the program can be used in more situations (so it is more robust), and so on. Notably, the triple notation defines the relation between program code and logic, and all of formal methods follows: one can refine a formal specification to a program, one can talk about invariants, assertions and so on with rigor and clarity.

We introduce the use of Hoare triples to help designers, developers and programmers to reason about user interfaces. Now note that the precondition  $P$  is a fact the user “knows” and the goal the user wishes to achieve is a postcondition  $R$ . More precisely what the user knows should imply  $P$ , and  $R$  should imply what the user wishes to achieve. In general the user will have to learn (mainly by experimenting with interactive systems) how to translate their tasks into sequences of  $Q$  to incrementally achieve subgoals that collectively achieve their tasks. There is a lot of complex human factors qualifying all those claims [18]—including the fact that the user may not often look at the display so will rely on keystroke rules alone [25]—but the converse can be expressed without qualification: if the designer does not know the triples, the user has no grounds for valid reasoning, and the user interface cannot be used dependably. It cannot be relied on to accomplish the intentions of its user [13].

User actions  $Q$  are simple but  $P$  and  $R$  are complex: Hoare’s notation then becomes hard to read as  $Q$ , often being just a single keystroke, gets lost in the details. Therefore we use an equivalent notation, inspired by the elegant visual layout of Z schemas [11], [32], but with a wavy line to avoid confusion with Z itself:

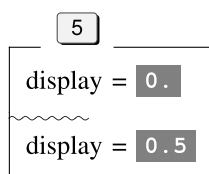


We will occasionally need local definitions to declare names and types, and we write these below the schema, e.g.:

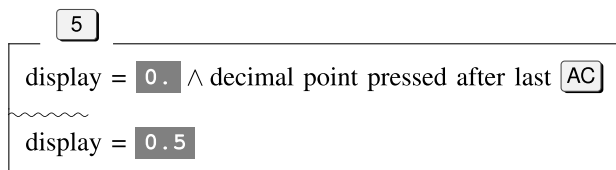


These definitions are not visible to the user: they allow us to write triples concisely.

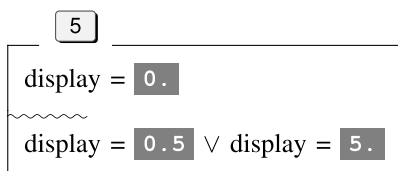
Next we show a very simple example, defining what happens when the user presses  $\boxed{5}$  when the display shows  $\boxed{0}$ :



Few devices behave like this. Instead, to remain faithful to what real devices typically do, we must tighten the precondition:



or relax the postcondition:



Complex preconditions or postconditions show how the notation highlights uncertainty that can cause confusion for users. In the example here, the confusion can easily be prevented by better design.

Most program code uses variables. Thus considering  $\text{max} = a[i]$  as an example: while the program code does not change, its meaning changes depending on the values of the variables  $a$  and  $i$ . In contrast, users *always* execute a concrete sequence of actions or commands with no variables. A user cannot do  $x$  as they have to do something specific; so a variable like  $x$  can be used in our notation to represent what a user could do. In particular, the user pressing  $x$  and the user pressing  $\boxed{X}$  are different—in the former case, the notation means that user presses *some* key, namely the key that is the value of  $x$ , and in the latter case, the notation means that the user presses the *specific* key  $\boxed{X}$  itself.<sup>3</sup>

In user interfaces, we are looking for the meanings of user actions such as  $\boxed{X}$  and programming language concepts like “scope” become concepts like “mode” and “window.” In the present paper, we focus on the mode of number entry. More general analysis must be left for further work—except to note that a reason for the success of object orientation is that programming scope becomes tightly related to modes that make sense to the user. For example, a *user interface field* in which the user enters numbers will also be a *program object* that encapsulates the implementation of user actions in that field: that is, the meaning of, say,  $a := b$  in the object determines the semantics of  $\boxed{X}$  in the field.

3. A user interface might be used to edit a computer program that has variables, or a web form might have fields whose values can be changed, or a user might refer to a knob as “variable”—but turning it cannot be variable, it has to be turned a specific angle. Thus *applications* may have variables, but user *actions* are never variables; they are always concrete instances.

We will often want to be more specific than “any key” as a user action  $Q$ . Typically we will write  $x \in \{0123456789\}$ , for example, requiring  $x$  to be any digit key. For convenience we define  $\text{numerickey} = \{0123456789\bullet\}$ , so a numeric key (as defined) is a decimal digit or a decimal point. More generally we could use Hartson’s User Action Notation (UAN) [12], but it would introduce a notational complexity beyond the needs of the present paper. The generality of UAN is not needed here, nor explained here, though if the notation used in this paper was implemented in a tool it would make sense to use such an existing standard.

We distinguish between mathematical variables, which are written in italics (like  $x, y, z$ ) and user interface properties (like Display, Error, On), which are capitalised and written in Roman. A mathematical variable anywhere in the triple  $\{P\}Q\{R\}$  denotes the same value everywhere in the triple. However, a user interface property mentioned in  $P$  means its state before the action  $Q$ , and mentioned in  $R$  means its state after the action. It would be counter-intuitive to refer to, for example, the display *after* the user’s action in a precondition *before* it has occurred, and our notation makes this complex idea impractical to express. Hence what might have been written using just a postcondition,  $\text{On}' = \neg\text{On}$  (meaning On is flipped by  $Q$ , say by pressing an  $\boxed{\text{On/Off}}$  button), has to be written as a precondition  $\text{On}_n = s$  and a postcondition  $\text{On} = \neg s$ .

Further conventions are familiar from programming language notations:

- ' $x$ ' means the literal symbol  $x$ . The notation generalises in the usual way: ' $abc$ ' means the sequence of symbols  $a$  then  $b$  then  $c$ .<sup>4</sup> We use the term *string* to be the type of a sequence of symbols, of any (natural number) length, including "" which is the string of length 0.
- On (i.e., written in Roman) are variables representing the state of persistent objects in the user interface.
- $x$  (i.e., written in italic) is a local variable representing a value used in the specification of a user action. The variable has no significance beyond of the scope of the specification.
- $|x|$  means the string  $x$  has this number of symbols; hence  $|\text{""}| = 0$  and  $|\text{"900"}| = 3$ . (In the Java program code above, the notation was  $x.\text{length}$ .)
- $\in \dots$  finally we take some liberties. Generally  $e \in S$  means the element  $e$  is in the set  $S$  or is of type  $S$  (a type can be thought of as the set of every possible value of that type), but we will use  $\in$  on collections that are not sets, such as strings.

Sequences of symbols ' $abc$ ' can either mean the user pressed these keys or that these symbols are displayed for the user to read. It is mnemonic to represent keys the user pressed as  $\boxed{a} \boxed{b} \boxed{c}$ , and symbols the user sees displayed as  $\boxed{abc}$ . Hence we may write  $\text{Display} = \boxed{3}$  as a way of writing  $\text{Display} = '3'$  or  $\text{Display} = \blacksquare$  meaning the display is initially blank (showing "", i.e., the empty string).

4. Invisible symbols, like tabs, and symbols such as ' are conventionally represented using backslash notations (e.g., '\ ') but this paper is not concerned with these lexical issues.

The display on a typical device for number entry will be composed of a numeric display (the main display) and various indicators, such as error flags. We will refer to these as Display, Error, etc, and treat them as variables; for example a precondition  $\text{Display} = d$  means it is true that the display is showing  $d$  before the user starts pressing keys.

In this paper we are not concerned with various ergonomic issues, even though they are clearly important; for example:

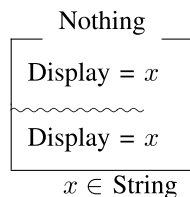
- The difference between zero, nothing and space is complicated. The display  $\blacksquare$  notation would be inappropriate if the user interface has a space key (or uses spaces instead of commas to separate digit groups). One might use the conventional representation of space, but it is unlikely a user would understand  $\blacksquare$  as “nothing”—without a symbol for nothing a user cannot distinguish a display that is off or broken from one that is on but displaying nothing. Some user interfaces blink zeros when they are showing “nothing.”
- We assume that if we write  $\text{Display} = d$  we mean the display shows  $d$  and the user actually sees  $d$ . In some cases, however, the display may be truncated or have some sort of scrolling feature so that the user sometimes sees a substring of  $d$ —the display  $\blacksquare 456$  may mean 123,456 or 456,789, or almost anything. We consider this unacceptable, but there are clearly conditions where showing less than  $d$  is unavoidable. Some ergonomically-designed cue should be used to indicate that there is additional information that is not displayed.
- The decimal point may be different on the keys and on the display (e.g.,  $\blacksquare$  and  $\blacksquare$  or  $\blacksquare$  in some countries).
- Decimal digits after a decimal point may be smaller [41].

## 6 RULES FOR COMMON DESIGN DECISIONS

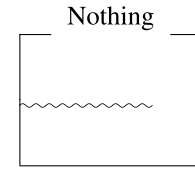
We now create rules that represent typical properties of number entry user interfaces. If we were developing or analysing an actual system we would create rules for each possible action. Here we will discuss representative rules for a range of real devices to explore what they say users must know and whether they may be poor design decisions as a result.

### 6.1 Doing Nothing—Time-Outs

If the user does nothing, then usually nothing happens. Our convention will be that if nothing changes we do not need to say so. Thus the following

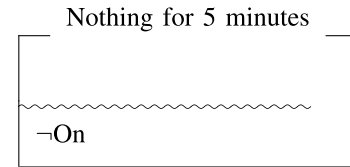


is unnecessarily cluttered, and is more satisfactorily represented by



which looks much more like a definition of doing nothing!

For some user interfaces, if the user does nothing for five minutes (or so) something happens, which might be expressed informally as:



In this example, the precondition is true under all circumstances so it does not need specifying explicitly, and it has been left blank. The postcondition  $\neg \text{On}$  means that after completion of the action, “nothing for 5 minutes,” it will be the case that the device is not on (On is false). Switching the device off ( $\neg \text{On}$ ) may be a safer choice than the design choice of the Graseby 3400 (Section 2.12), where the device remains on but the number displayed is set to zero without warning the user.

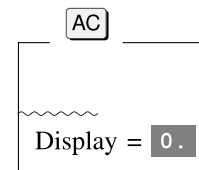
While we do not think time-outs are necessarily a good idea, not reasoning about them and their applicability to the domain the number entry is intended for is worse; here, the time-out has an explicit rule designers can consider carefully.

### 6.2 Clear Rule

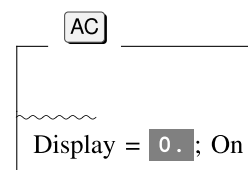
The next simplest rule is that pressing the All Clear key  $\boxed{\text{AC}}$  clears the display; on most calculators, pressing  $\boxed{\text{AC}}$  makes the display show  $\boxed{0.}$ . Hence

$$\{\text{true}\} \boxed{\text{AC}} \{ \text{Display} = \boxed{0.} \}$$

or in our notation (and simplifying the true precondition):



In fact,  $\boxed{\text{AC}}$  also switches the device on—perhaps it seems obvious that if the display shows something the device is on, but we should make it clear:





The semicolon above is an useful way of writing “and” with a low operator precedence; had  $\wedge$  been used, brackets would have been needed to write  $(\text{Display} = 0.) \wedge \text{On}$ . The semicolon does not mean the display shows 0. then the calculator is on; it means that after  $\boxed{\text{AC}}$  has been pressed, then the post conditions—the display shows 0. and the calculator is on are both true.

### 6.3 Basic Append Rule

If the user presses a numeric key  $x$  (a digit or decimal point) we would expect the key to appear in the display. Expressed more formally:

$$\frac{x \in \text{numerickey} \quad \text{---}}{\text{Display} = d \quad \text{---}} \quad \text{---}$$


---


$$\text{Display} = dx$$

This simple rule does not capture what most devices do. If the device is off, then it will display nothing and after pressing  $x$  it will still display nothing. Hence:

$$\frac{x \in \text{numerickey} \quad \text{---}}{\text{On}; \text{Display} = d \quad \text{---}} \quad \text{---}$$


---


$$\text{Display} = dx$$

This is still incomplete. If the display is full when it shows  $d$ , it cannot show  $d$  and  $x$  together. Let us suppose  $\text{max}$  is the maximum capacity of the display (for example,  $\text{max} = 8$  characters on the HS-8V), then we can consider two possible solutions for the rule:

(a.1) 
$$\frac{x \in \text{numerickey} \quad \text{---}}{\text{On}; \text{Display} = d; |d| < \text{max} \quad \text{---}} \quad \text{---}$$

---


$$\text{Display} = dx$$

(a.2) 
$$\frac{x \in \text{numerickey} \quad \text{---}}{\text{On}; \text{Display} = d; |d| \geq \text{max} \quad \text{---}} \quad \text{---}$$

---

or

(b) 
$$\frac{x \in \text{numerickey} \quad \text{---}}{\text{On}; \text{Display} = d \quad \text{---}} \quad \text{---}$$

---


$$\text{Display} = \begin{cases} |d| < \text{max} : dx \\ |d| \geq \text{max} : d \end{cases}$$

In form (a), two rules are required, whereas (b) combines the two rules into a single rule where the ambiguity is more

obvious. The bracket notation is syntactic sugar:

$$\left\{ \begin{array}{l} a : b \\ c : d \\ \dots \end{array} \right. \equiv (a \wedge b) \vee (c \wedge d) \vee \dots$$

The point of the schemas is to help us reason clearly about a user interface, analogously to how a user would think—certainly if we cannot express our thoughts precisely, the user will not be able to! In particular, if a user wishes to think clearly, the notation captures everything that is in principle relevant to their reasoning. We notice that in case (a) we are assuming the user knows whether the display is full before pressing a key; we suspect that is unlikely. Case (b) is preferred as it makes clear that when the user presses  $x$ , there may be either of two outcomes. Whether these are desirable outcomes we will return to in a moment.

We ignored that on many displays the decimal point occupies no extra space. On the EasyCalc, the seven segment display is large enough for 12 digits and 12 decimal points, one per digit (though it displays only one decimal point at any given time). If we wanted to be precise about the size of the display and the ability to include an “extra” decimal point, instead of using the notation  $|d|$  we should define a function like  $\text{width}(d)$ . This would also be useful for displays that use variable-width digit fonts (e.g., where 1 is narrower than 2).

### 6.4 Numeric Append Rule and Ambiguity

A number entry user interface displays numeric values, and the append rule described above is naïve. For example in the special case that the initial display is the 0, we have

$$\frac{x \in \text{numerickey} \quad \text{---}}{\text{On}; \text{Display} = 0 \quad \text{---}} \quad \text{---}$$


---


$$\text{Display} = x$$

This is correct even in the special case  $x$  is 0, though if  $x = \cdot$  then the final display would probably be 0. As we explained above, assumptions in the precondition are awkward. The rule is complex and better expressed as follows:

(c) 
$$\frac{x \in \text{numerickey} \quad \text{---}}{\text{On}; \text{Display} = d \quad \text{---}} \quad \text{---}$$

---


$$\text{Display} = \begin{cases} |z| \leq \text{max} : z \\ |z| > \text{max} : d \end{cases}$$

**where**  $z = \text{canonicalise}(d, x)$

Here  $\text{canonicalise}(\text{string}, \text{keypress})$  is a function that takes a displayed string and a key press and yields a string representing the canonical numerical value of its argument. Here are some examples of its behaviour:

- $\text{canonicalise}(0, 0) = 0$
- $\text{canonicalise}(0, 1) = 1$
- $\text{canonicalise}(0.00, 0) = 0.000$
- $\text{canonicalise}(12, 3) = 123$

The function `canonicalise` is not only a function that could be implemented as some program code but it also represents rules in the user model. The user models the device as “if the display is  $x$  and I press  $y$  then the display will become `canonicalise(x,y)`.” Of course the user model won’t be expressed in such words, but the meaning will be—or should be—equivalent.

What does `canonicalise` do with repeated decimal points? Many calculators ignore extra decimal points, so we have cases like:

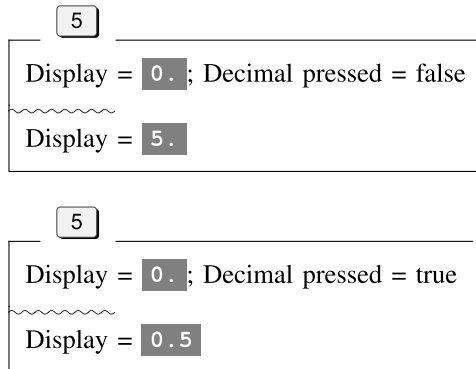
- `canonicalise(1.2, .)` = 1.2

Unfortunately the display 0. is ambiguous; we do not know, for instance, whether

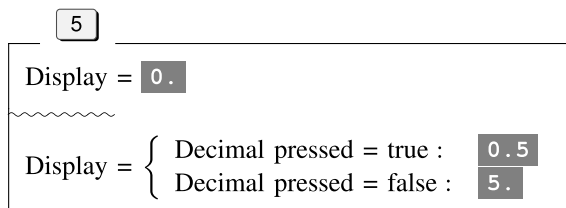
- `canonicalise(0., 0)` =  $\begin{cases} 0.0 \\ 0. \end{cases}$

In other words, `canonicalise` is not functional. Being non-functional means that what `canonicalise` does depends on more than its parameters (the display and the key pressed)—in other words, it becomes non-deterministic or unpredictable. Therefore the triple (c) above needs correcting.

The reason the ambiguity occurs is that when Display = 0. the user cannot tell whether . has already been pressed or not. If a decimal point has been pressed, the next digit is a fractional decimal digit, whereas if the decimal point has not been pressed yet, the next digit will be a units digit. The following two triples make this clear:



As mentioned above, hiding ambiguity in the preconditions is poor practice, not least because it creates two rules for one user action in this case;<sup>5</sup> a clearer formalisation is as follows:



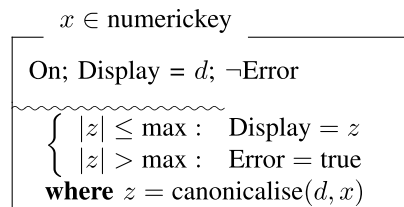
Now we have one rule, and the now obvious choice in the postcondition highlights a problem for a user. Most

5. In fact, if written in this style, there are lots of rules for 5 because these are only the two rules for when the display is 0., and they say nothing about what happens when it displays other values.

calculators always display a decimal point, which is the cause of this ambiguity. Ambiguity is bad [24], and it is encouraging to see how easy it is to avoid in this case. There is an obvious solution: do not display a decimal point when one has not been pressed. If we do this, Display = 0. implies the decimal point has been pressed, and hence the condition “decimal pressed” is true, and conversely when Display = 0 then the condition “decimal pressed” is false.

### 6.5 Persistent Error

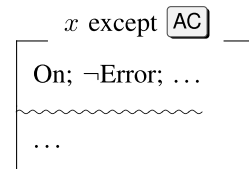
The HS-8V and the EasyCalc behave differently when the display is full. When the display is full on the HS-8V, further keystrokes are ignored and there is no error; on the EasyCalc E is displayed to indicate an error. We can represent this thus:



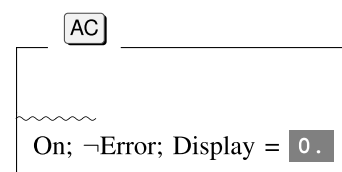
The EasyCalc is somewhat more sophisticated than this: if a decimal point has been entered, then the excess digits (or further decimal points for that matter) are treated as “insignificant” and ignored (as would happen on the HS-8V too) but if a decimal point has not been entered, an error occurs since discarding a digit would display a number that was wrong by a factor of about 10.

We could write either `Error = true` or `Error = E`, etc., meaning more specifically that a region of the display reserved for error notifications is displaying E. We prefer to use the logic form as it does not presuppose a particular way of representing errors to the user (E or Error etc.), and it allows the variable Error to appear in logical expressions directly without referring to the concrete choice of warning.

Now we have introduced Error in the modelling, all previous rules for the EasyCalc need modifying:



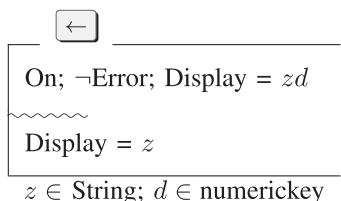
and the rule for AC more specifically becomes:



So on the EasyCalc, when an error is detected, the user is warned and the warning is persistent until the user clears the error condition by pressing AC. Or so it seems ...

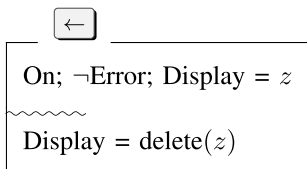
### 6.6 Delete Rule

The EasyCalc has a delete key, , which deletes numeric keys. One would imagine its behaviour is as follows:







Notice that in this triple we had to specify the types of  $z$  and  $d$  since they cannot be inferred from the context. The triple does not specify what happens when Display cannot be partitioned as  $zd$ —which happens when the display is showing nothing—but in fact if the EasyCalc is switched on it always can be.

However, the EasyCalc does not work like this: the delete key ignores the decimal point. Its definition is therefore more complex, and at a first attempt might be written as follows:

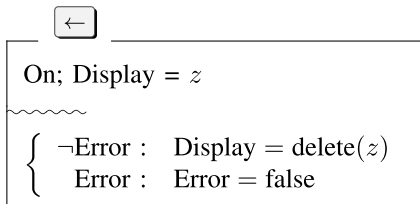



We illustrate the behaviour of the EasyCalc delete() with a few cases:

- delete(0.) = 0.
- delete(1.) = 0.
- delete(1.2) = 1.
- delete(23.) = 2.

This quirky behaviour has the result that a sequence of keystrokes  $xd$   for any digit  $d$  will be equivalent to  $x$  (i.e., the single digit  $d$  was deleted), as expected, but that any sequence equivalent to  $xd$    will be equivalent to  $x$  too. In other words, the delete key deletes more than the last keystroke if the last keystroke was a decimal point. If the user tries to correct the slip of keying  $n = 2$  decimal points instead of the single one intended by pressing  once, the preceding digit will disappear! Even pressing a single decimal point in error cannot be corrected. Ironically, the delete key is there to *correct* errors, not add to them.

Delete on the EasyCalc is quirky in another way too. If the display is full, then the delete key resets the error—however, others types of error are not reset by the delete key.



The EasyCalc does not count how many excess keystrokes the user keyed. So for example if the user keyed 15 keystrokes (much larger than  $\max = 12$ ) then a *single*  is sufficient to

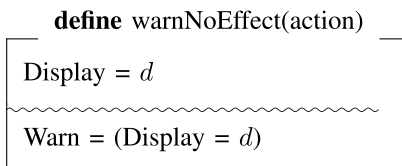
clear the error—yet strictly the user should have pressed delete at least  $3 = 15 - 12$  times to clear the error.

### 6.7 Rules for Consistency

While the schemas bring out the meanings of individual user actions, they have the disadvantage that they do not help describe consistent features across an interface. We might want error handling to be consistent, but if it is repeated in every schema then there is a danger that clerical errors will slip in.

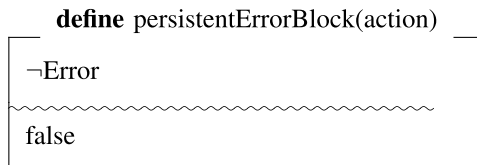
There are two solutions, to use theorem proving tools (to establish the consistency properties) or to use abstraction. We have already used abstraction in using features like the function canonicalise: it appears in many places but in each case has the same meaning. Abstraction introduces named features that can be instantiated in multiple schemas.

If a user performs any action, presumably they want an effect, or possibly the action was in error (say, pressing a letter key during number entry) and they want assurance there was no effect. Instead of repeating this rule in many schemas, it could be stated once:




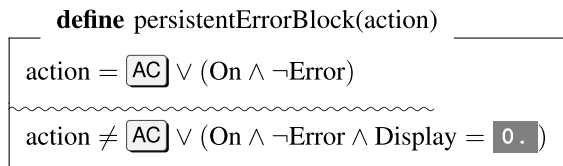
This says Warn is true if the user's action does not change the display.

Earlier (Section 6.5) we proposed persistent errors: when an error flag is set, actions are inhibited. Whatever choice is made, it should be consistent. For example:

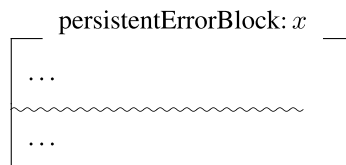


This schema also asserts that if a device is off nothing happens anyway, regardless of whether there is an error.

In Section 6.5 there were two rules for errors, one for  and one for all other keys. Instead, the rules can be combined:



In contrast to a definition of a function like canonicalise, this definition has pre and post-conditions. The rule can now be applied to any action:



The definition  $\{P_a\}\mathbf{define} Q_a(x)\{R_a\}$  applied in a triple  $\{P\}Q_a : Q\{R\}$  means  $\{P \wedge P_a\}Q\{R \wedge R_a\}$ , with the usual renaming of  $x$  as  $Q$  within  $P_a$  and  $R_a$ .

## 6.8 A Rule for Sequence

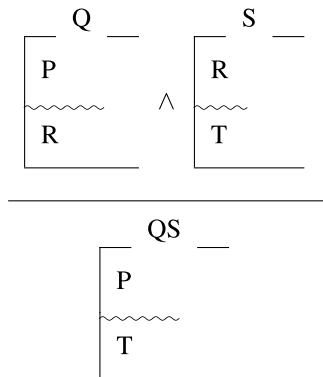
In many programming languages, the semicolon separates statements that are executed in sequence. The Hoare triple for it,  $\{P\}Q;S\{T\}$ , follows from the premises  $\{P\}Q\{R\}$  and  $\{R\}S\{T\}$ , sharing the midterm  $R$ . The rule of inference for semicolon is written out in standard form as:

$$\frac{\{P\}Q\{R\} \wedge \{R\}S\{T\}}{\{P\}Q;S\{T\}}$$

Now consider a program

```
readAndDo(Q); readAndDo(S)
```

which uses the standard `;` operator to implement a program enabling a user doing  $Q$  followed by doing  $S$ . Hence, the rule for the meaning of the user doing  $Q$  then  $S$  must be



This rule states that a properly specified user interface remains properly specified as the user performs a sequence of actions.

## 7 DISCUSSION

Section 3 argued that it is deceptively easy to write program code to find the maximum value of an array. Bugs in program code can be avoided using formal methods, a familiar point that has been widely presented in the literature (e.g., [6]), however the advantages are only achieved if we have formal requirements. For maximum the requirements are so obvious we did not define maximum! But in user interface design, the requirements are often implicit, complex, and partially unknown. Unsurprisingly, there are numerous bugs and inconsistencies in user interfaces (Section 2), for the same reasons as any other bugs in programs (Section 3)—lack of clear requirements combined with lack of formal reasoning. Section 5 then showed how formal reasoning can be used in user interface design, hence helping avoid bugs. The formalisation made requirements and trade-offs between requirements explicit, but it left begging the question what requirements do we really want?

### 7.1 What Do We Want?

User interfaces seem simple because one cannot see everything that can happen: this is a problem for users, designers and programmers. It is not clear we really know what we

want to do, nor that we can reliably implement it. Often users cannot articulate exactly what they want to do, and even if they did, it might be different from what they need since many properties of human behaviour are unavailable to consciousness.

We were critical of user interface design defects because we claimed they were obviously the consequences of poor programming practices, needing but showing little evidence of formal reasoning. This is a superficial stance. We could equally have tried to formally specify what numeric user interfaces *actually* do, and then we could have presented these specifications as correctly implemented. Indeed, formal methods has no value system it imposes: it does not judge what the right system is, merely that if you decide what the right system is you can more reliably obtain the system you wanted. It is therefore useful to distinguish between epistemology (knowing what we want to do) and logic (knowing we are reasoning correctly about what we want to do) [31]. Indeed Aristotle would go further: knowledge is only useful if we act on it. Hence, how do we persuade others to do good (rhetoric) and how do we act appropriately in the communities of designers and developers (politics) so better user interfaces are designed and manufactured (and out-sell the worse ones)? These critical topics build on the foundation of formal methods, reliable reasoning about user interfaces.

Evidently, our discussion glossed the value system. We took it as self-evident and not needing elaboration that a user interface for number entry should be predictable. Having made that value judgment, we can then refine it into some logical framework, then design user interfaces that are predictable in the chosen sense. A formal methods approach then facilitates this second process: correctly implementing what one wants to implement.

What, then, is predictability? We have discussed predictability and its variations at length elsewhere and successfully linked it to formal reasoning [11], [21], [22], [34]. For our present purposes we can summarise predictability informally [33]:

*Predictability:* A user can successfully use the system with their eyes shut right until the moment they want answers.

This simple formulation is consistent with eye tracking experiments [25]: users infrequently fixate on (look at) the display because they need to fixate on finding and pressing the right keys. Effectively, their “eyes are shut” in terms of reading information on the display.

If the user thinks “**X** does something,” it should always do that thing; otherwise they would have to open their eyes to see the difference. In other words the user interface has no modes that change the meaning of the user’s actions, and there must be features like a key **C** that completely resets the user interface so that the user can start fresh without having to read the display. Realistically, we also know users will make slips, occasionally pressing the wrong keys. When they press the wrong key, they will want to correct what they have done. Thus, the delete and clear keys must be predictable, and not depend on the last or previous key-strokes (e.g., whether they were decimal points or not). If the user makes a slip that they do not notice, then the device should (if possible) keep track of the error until the user



metaphorically opens their eyes. This form of predictability favours the string interpretation of number entry user interfaces (Section 2.1).

Predictability does not apply only to isolated devices: predictability is with reference to what the user knows, and what the user has learned from use of other related systems. We need to reduce *transfer errors*, which occur when a user performs the right actions on the wrong device. We should be developing a new, more dependable and predictable user interface standard. Perhaps there should be a conformity certificate or badge with all new, improved user interfaces? The certificate or other identifying markings should be indelible so that they are present not just during purchase or procurement, but to reassure users any time in the future life of the product [44]. Ideally, in critical domains, only certified improved user interfaces should be used.

## 7.2 Completeness of Requirements

Always, the completeness of what we want must be questioned. Requirements and specification may seem correct and consistent, but there are often additional factors that have not been considered—“unknown unknowns.” They will be implemented following arbitrary and often unnoticed choices. For example, on many devices, when the delete key is pressed the display content visually moves right. In the special case that all digits displayed are the same, “moving right” is visually indistinguishable from deleting the left-most digit, which is unfortunate because the right-most digit has been deleted. This confusion is eliminated by left-justifying the display, or by animating it moving right (which is not possible on seven segment displays). Indeed, the Hewlett-Packard 20S (produced 1988-2003) had a left-justified display and a delete key, so there is a precedent. The point is that a design choice can be made with no representation in the program or requirements. Too often design decisions are justified after the fact for no reason better than avoiding the cost of improving them.

When a program fails to work this cannot be denied: the program code must be wrong; but when a user interface fails there is a temptation to say the user is wrong, then the program behind the user interface does not need correcting, since the user needs to learn how to use it properly. Neither users nor designers want to be told they are “wrong” and it is easy to see that a culture of denial arises. Moreover, people do not make mistakes they notice—mistakes happen because the errors are unnoticed: all of us are therefore very weakly aware of problems with user interfaces.

Another reason for denial is that finding user interface bugs is tedious, and few users (or empirical experiments) are persistent enough to uncover bugs thoroughly—and when bugs are found they may be hard to notice and are certainly hard to reproduce from memory. Often users think bugs are their own fault and having problems with computers is embarrassing, so bugs are under-reported and hence requirements persist in being incomplete.

## 7.3 Misconceptions of Usability

After just programming and not thinking about user interface properties, the next most common problem is confusing speed, error tolerance and flexibility for usability. For

example, allowing a user to change the sign of a number anywhere might appear to be more usable than a more restrictive approach. However, occasionally, the user (and, as we have seen, the programmer too) will get confused and the consequence is an error which might result in a catastrophe. The minor delay treating change sign properly is negligible compared to the delay of sorting out a catastrophe. In other words, usability has to be seen in a larger context: speeding up number entry should not be confused with usability.

## 7.4 The Need for Experiments and Standards

The HR-150TEC has a double zero key  $\boxed{00}$  that *probably* speeds up number entry. It reduces keystrokes needed for numbers with consecutive zeros, but it slows down the user because there are more keys to choose from and the user has to be more careful to press the correct key from the larger number of keys. It also introduces a new uncertainty: what does  $\boxed{00}$   $\boxed{\leftarrow}$  do? Potentially, correcting errors is so slow that on average any gain is lost; we do not know.

The HR-150TEC implements delete as deleting digits ignoring decimal points, so  $\boxed{00}$   $\boxed{\leftarrow}$  =  $\boxed{0}$  (except when the display is too small to display all the zeros). I happen to think this is wrong, but the HR-150TEC is marketed to accountants, and I am not an accountant and I have insufficient insight into how they expect numbers to work. One should do experiments to establish how the intended users actually work: find out which design lowers errors; secondly, establish whether the potential confusion of a feature warrants removing the feature from the design. One should also conform to standards: for medical devices, placing  $\boxed{0}$  next to  $\boxed{\bullet}$  is known to be a bad design choice [10]; the HR-150TEC places  $\boxed{00}$  next to it (see Fig. 2 and Section 2.13), and may be a worse decision.

## 8 CONCLUSIONS

User interfaces for number entry present a confusing variety of inconsistent design decisions, even across models from the same manufacturer. One imagines that user interface design for number entry is thought to be so easy that it is “just” programmed, and what happens happens without further thought.

User interface design has long emphasised “user centred design” where improvements are sought through experiments with users [20]. Our example of number entry shows that for at least 30 years, user experiments with many number entry systems have failed to identify easily-fixed defects.

- User interface design—HCI, human computer interaction—needs to mature and include formal methods in its armoury of tools.

This paper introduced a notation to help do this. Moreover, the solutions suggested here can be implemented with little disruption, little more than upgrading firmware.

There is nothing special about number entry, other than frequently occurring in safety critical applications. Number entry seems simple, but few user interfaces manage to implement it well, even though the syntax for Arabic numerals is theoretically sorted out. Many other types of user interface, from TV remote controls to spreadsheets, from wifi to document processing, all have defective user interfaces, but their bugs are harder to articulate and

perhaps much harder to reach consensus over: it is easy to say that  $\boxed{2}$   $\boxed{7}$   $\boxed{\bullet}$   $\boxed{5}$  should behave like 27.5, but it is much more tedious to write down rules for a user's wifi configuration. The user interface should not be ignored by formal methods.

- Formal methods needs to develop notations and tools to help specify and manage user interaction.

If we do this, and in particular design out errors users are unlikely to notice, then we will get closer to Hoare's vision, "it will be possible to place great reliance on the results of the program" [13].

## APPENDIX

### Brief Description of User Interface Models

In addition to common PC user interfaces, a variety of devices were referenced in the body of the paper. All devices discussed in this paper, summarised in the table below, have number entry user interfaces with numeric keys very similar to that shown in Fig. 1, except the BBraun Infusomat (which has four arrow keys) and the GE Dash 4,000 (which has a knob).

Infusion pumps and syringe drivers are medical devices used for automatically delivering drugs to patients. They may contain calculators to calculate doses and delivery rates. A syringe driver holds a syringe whereas an infusion pump is typically used to deliver drugs from a bag. A syringe driver typically knows the length, diameter and make of the syringe and possibly the drug itself, whereas with an infusion pump the drug bag is separate, so typically an infusion pump only knows the rate of flow, not the volume or drug.

Device	Type
Abbott Gemstar	Infusion pump
Abbott AimPlus	Infusion pump
Alaris PC	Infusion pump
Apple iPhone	Smart phone (touch screen)
Baxter Colleague 3	Infusion pump
BBraun Infusomat	Infusion pump (arrow keys)
BBraun Vista Basic	Infusion pump
Canon F-502G	Calculator
Casio DJ-120D	Calculator
Casio fx-85GT	Calculator
Casio HR-150TEC	Calculator (paper roll record)
Casio HS-8V	Calculator
Casio MU-120T	Calculator
Casio OfficeCalc 100	Calculator
CME BodyGuard 545	Infusion pump
DRE Avanti Plus	Infusion pump
DRE SP1500 Plus	Syringe driver
GE Dash 4000	Patient monitoring system (knob)
Graseby 500	Infusion pump
Graseby 3400	Syringe driver
Graseby Omnifuse	Syringe driver
HP 20S	Calculator
HP EasyCalc 100	Calculator
HP SmartCalc 300s	Calculator
Samsung Android	Tablet (touch screen)
Sigma 6000 Plus	Infusion pump
Sigma 8000 Plus	Infusion pump
Sigma Spectrum	Infusion pump
SK Medical SK-500III	Syringe driver
SK Medical SK-600III	Infusion pump
Upreal UPR-900	Infusion pump
Upreal CTN-TCI-V	Syringe driver

## ACKNOWLEDGMENTS

This research was funded by the United Kingdom Engineering and Physical Sciences Research Council (EPSRC) Grant numbers [EP/G059063, EP/K504002, EP/L019272/1]. Paul Cairns, Abigail Cauchi, Michael Harrison, Paolo Masci, Gordon Pace and Richard Young all made many very valuable comments for which the author is grateful. The Medical Device PnP group at Massachusetts General provided laboratory facilities for which we are grateful.

## REFERENCES

- [1] M. G. A. Ament, A. L. Cox, A. E. Blandford, and D. P. Brumby, "Making a task difficult: Evidence that device-oriented steps are effortful and error-prone," *J. Exp. Psychol.: Appl.*, vol. 19, no. 3, pp. 195–204, 2013.
- [2] P. Cairns, M. Jones, and H. Thimbleby, "Usability analysis with Markov models," *ACM Trans. Comput.-Human Interaction*, vol. 8, no. 2, pp. 99–132, 2001.
- [3] P. Cairns and H. Thimbleby, "Reducing number entry errors: Solving a widespread, serious problem," *J. Roy. Soc. Interface*, vol. 7, no. 51, pp. 1429–1439, 2010.
- [4] A. Cauchi, A. Gimblett, P. Curzon, P. Masci, and H. Thimbleby, "Safer '5-key' number entry user interfaces using differential formal analysis," in *Proc. BCS Conf. Human-Comput. Interaction*, 2012, vol. 26, pp. 29–38.
- [5] R. L. Deininger, "Human factors engineering studies of the design and use of pushbutton telephone sets," *Bell Syst. Tech. J.*, vol. 39, no. 4, pp. 235–255, 1960.
- [6] E. W. Dijkstra, *A Discipline of Programming*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1976.
- [7] K. Fu, "Trustworthy medical device software," in *Public Health Effectiveness of the FDA 510(k) Clearance Process*, Inst. Medi., Nat. Academies Press, 2011.
- [8] S. W. Gilroy and M. D. Harrison, *Interactive Systems, Design Specification, and Verification: 12th International Workshop, Lecture Notes in Computer Science*, Springer-Verlag New York, 2005.
- [9] F. G. Halasz and T. P. Moran, "Mental models and problem solving in using a calculator," in *Proc. ACM SIGCHI Conf. Human Factors Comput. Syst.*, 1983, pp. 212–216.
- [10] S. Halls, *Design for Patient Safety: A Guide to the Design of Electronic Infusion Devices*. National Patient Safety Agency, 2010.
- [11] M. D. Harrison and H. Thimbleby, *Formal Methods in Human Computer Interaction*. Cambridge, U.K.: Cambridge Univ. Press, 1990.
- [12] H. R. Hartson and P. D. Gray, "Temporal aspects of tasks in the user action notation," *Human-Computer Interaction*, vol. 7, no. 1, pp. 1–45, 1992.
- [13] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580 & 583, 1969.
- [14] International Standards Organization, *Ergonomics of Human-System Interaction—Part 210: Human-Centred Design for Interactive Systems*, ISO 9241–210, 1st ed., 2010.
- [15] Institute for Safe Medication Practices. (2006, Sep.). ALERT: Potential for 'Key Bounce' with infusion pumps. *ISMP Canada Safety Bull.* [Online]. 6(6). Available: [www.ismp-canada.org](http://www.ismp-canada.org)
- [16] Institute for Safe Medication Practices. (2007). *Fluorouracil Incident Root Cause Analysis* [Online]. Available: [www.ismp-canada.org](http://www.ismp-canada.org)
- [17] Institute for Safe Medication Practices, *List of Error-prone Abbreviations, Symbols and Dose Designations* [Online]. Available: [www.ismp.org/tools/abbreviations](http://www.ismp.org/tools/abbreviations)
- [18] P. N. Johnson-Laird, *Human and Machine Thinking*. Hillsdale, NJ, USA: Lawrence Erlbaum Assoc., 1993.
- [19] J. C. Knight and N. G. Leveson, "A reply to the criticisms of the Knight & Leveson experiment," *SIGSOFT Softw. Eng. Notes*, vol. 15, pp. 24–35, 1990.
- [20] T. K. Landauer, *The Trouble with Computers*. Cambridge, MA, USA: MIT Press, 1995.
- [21] P. Masci, R. Ruksenas, P. Oladimeji, A. Cauchi, A. Gimblett, Y. Li, P. Curzon, and H. Thimbleby, "The benefits of formalising interactive number entry case studies with drug infusion pumps," in *Innovations in Systems and Software Engineering*, London, U.K.: Springer-Verlag, 2013, pp. 1–21.

- [22] P. Masci, Y. Zhang, P. Jones, P. Curzon, and H. Thimbleby, "Formal verification of medical device user interfaces using PVS," *Proc. 17th Int. Conf. Fundamental Approaches Soft. Eng. FASE'14*, doi: 10.1007/9783642548048\_14, 2014.
- [23] R. E. Mayer and P. Bayman, "Psychology of calculator languages: A framework for describing differences in users' knowledge," *Commun. ACM*, vol. 24, no. 8, pp. 511–520, 1981.
- [24] D. A. Norman, "Design rules based on analyses of human error," *Commun. ACM*, vol. 26, no. 4, pp. 254–258, 1983.
- [25] P. Oladimeji, A. Cox, and H. Thimbleby, "Number entry interfaces and their effects on errors and number perception," in *Proc. IFIP Conf. Human-Comput. Interaction*, 2011, pp. 178–185.
- [26] K. A. Olsen, "The \$100,000 keying error," *IEEE Comput.*, vol. 41, no. 4, pp. 1005–108, Apr. 2008.
- [27] F. Paternò, C. Santoro, and J. Ziegler, eds., *Proc. ACM SIGCHI Symp. Eng. Interactive Comput. Syst.*, ACM, New York, NY, USA, 2014.
- [28] K. R. Popper, *Conjectures and Refutations: The Growth of Scientific Knowledge*, 2nd ed. Evanston, IL, USA: Routledge, 2002.
- [29] J. Reason, *Human Error*. Cambridge, U.K. Cambridge Univ. Press, 1990.
- [30] C. Runciman and H. Thimbleby, "Equal opportunity interactive systems," *Int. J. Man-Mach. Stud.*, vol. 25, no. 4, pp. 439–451, 1986.
- [31] J. Rushby, "Logic and epistemology in safety cases," in *Proc. 32nd Int. Conf. Comput. Safety, Rel. Security*, 2013, pp. 1–7.
- [32] J. M. Spivey, *Z. Notation: A Reference Manual*, Prentice Hall International series in Computer Science. Englewood Cliffs, NJ, USA: Prentice-Hall, 1988.
- [33] H. Thimbleby, "Guidelines for 'manipulative' text editing," *Behav. Inf. Technol.*, vol. 2, no. 2, pp. 127–161, 1983.
- [34] H. Thimbleby, *User Interface Design*. Reading, MA, USA: Addison-Wesley, 1990.
- [35] H. Thimbleby, "A new calculator and why it is necessary," *Comput. J.*, vol. 38, no. 6, pp. 418–433, 1995.
- [36] H. Thimbleby, "Calculators are needlessly bad," *Int. J. Human-Comput. Stud.*, vol. 52, no. 6, pp. 1031–1069, 2000.
- [37] H. Thimbleby, "Permissive user interfaces," *Int. J. Human-Comput. Stud.*, vol. 54, no. 3, pp. 333–350, 2001.
- [38] H. Thimbleby, "Interaction walkthrough: Evaluation of safety critical interactive systems," in *Proc. 13th Int. Conf. Des., Specification, Verification Interactive Syst.*, 2007, pp. 52–66.
- [39] H. Thimbleby, *Press On*. Cambridge, MA, USA: MIT Press, 2007.
- [40] H. Thimbleby, "Heedless programming: Ignoring detectable error is a widespread hazard," *Softw.—Prac. Exp.*, vol. 42, no. 11, pp. 1393–1407, 2012.
- [41] H. Thimbleby, "Reasons to question seven segment displays," in *Proc. ACM Conf. Comput.-Human Interaction*, 2013, pp. 1431–1440.
- [42] H. Thimbleby, "Improving safety in medical devices and systems," in *Proc. IEEE Int. Conf. Healthcare Informat.*, pp. 1–13.
- [43] H. Thimbleby and A. Gimblett, "Dependable keyed data entry for interactive systems," *Electronic Commun. EASST*, vol. 45, pp. 1/16–16/16, 2011.
- [44] H. Thimbleby, A. Lewis, and J. G. Willians, "Making healthcare safer by understanding, designing and buying better IT," *Clinical Medicine*, 2015 (in press).
- [45] W. Thimbleby, "A novel pen-based calculator and its evaluation," in *Proc. 3rd Nordic Conf. Human-Comput. Interaction*, 2004, pp. 445–448.
- [46] W. Thimbleby, and H. Thimbleby, "Mathematical mathematical user interfaces," in *Proc. Eng. Interactive Comput. Syst.*, 2008, pp. 519–535.
- [47] R. M. Young, "The machine inside the machine: Users' models of pocket calculators," *Int. J. Man-Mach. Stud.*, vol. 15, no. 1, pp. 51–85, 1981.



**Harold Thimbleby** PhD, CEng, FIET, FLSW, FRCP (Edinburgh), Hon. FRSA, Hon. FRCP is at Swansea University, Wales. He gained his PhD degree in 1981. His research focuses on human error and computer system design, particularly for healthcare. In addition to over 388 peer reviewed publications, he has written several books, including *Press On* (MIT Press, 2007), which received the American Association of Publishers best book in Computer Science Award.

He received the British Computer Society Wilkes Medal. He is emeritus Gresham professor of Geometry (a chair founded in 1597), and has been a Royal Society-Leverhulme Trust senior research fellow and a Royal Society-Wolfson Research Merit Award holder. He has been a member of the United Kingdom Engineering and Physical Sciences (EPSRC) research council Peer Review College since 1994. See his website, [www.harold.thimbleby.net](http://www.harold.thimbleby.net), for more details.

▷ **For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/publications/dlib](http://www.computer.org/publications/dlib).**