UNIVERSITY *of* York

This is a repository copy of *From premature semantics to mature interaction programming*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/129124/

Version: Submitted Version

Book Section:

Cairns, Paul Antony orcid.org/0000-0002-6508-372X and Thimbleby, Harold (2018) From premature semantics to mature interaction programming. In: Howes, Andrew, (ed.) Computational Interaction. Oxford University Press, pp. 213-248.

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

From premature semantics
to mature interaction programming

Paul Cairns & Harold Thimbleby

June 7, 2017

Abstract

As HCI has progressed as a discipline, perhaps just as time has passed, the engineering work of programming has become increasingly separated from the HCI, the core user interface design work. At the same time, the sophistication of digital devices, across multiple dimensions, has grown exponentially. The result is that HCI and User Experience (UX) professionals and programmers now work in very different worlds. This separation causes problems for users: the UX is attractive but the program is unreliable, or the program is reliable but unattractive or unhelpful to use, correctly implementing the wrong thing.

In this chapter, we dig down from this high-level view to get to what we identify as a new sort of fundamental problem, one we call **premature semantics**.

Premature semantics must be recognised and understood by name by UX and HCI practitioners and addressed by programmers.

1   Introduction

It is a truism that the only reason computers exist is for humans to use them — computers are not a natural phenomenon that "just exist" (like rocks, flowers or cats) but they and what they do are made by people *for* people. Better computers are easier to use.

Somehow the two concerns:

- Building computers and how to understand computers — programming, computer science

- How computers are used and how to understand how they are used — user centred design, human factors

have become different specialities with few overlapping interests. University courses may study one and not the other. Web authoring systems mean that UX people work on web site design and user experience, but the underlying services are done by completely separate teams of programmers, as are the tools that the UX people use. This separation misses out a lot of beneficial cross-fertilisation.

We believe that this disciplinary separation is premature and unnecessary. What is taught in user interface design or human-computer interaction courses raises deep computational problems. Conversely, what is taught as basic theoretical computer science has applications directly in user interface design. After all, a programmer understanding computers and getting them to do things correctly for them is an almost identical problem to users understanding computers and getting them to do things correctly for them. Pretending that one is about people and the other about computers over-simplifies.

In this chapter we explore the split and we re-integrate basic computer science into the user centred design arena.

2 The trouble with programming

We all recognise that user interfaces are not always wonderful, and in safety critical areas like aviation what in hindsight look like quite simple design problems that should have been avoided have led to death.

The conventional perspective is that computer software is built by programmers who do not really understand the needs of actual users and what they really do to achieve their tasks. Landauer [9] is a classic exposition of this point of view; many have characterised the differences of opinion as "trench warfare" — with programmers as the enemy [5, 11, etc]. Although there are some problems of communication between programmers and user-centred designers [15], it is not that simple.

In the years since HCI became established as a research field (and usability and UX as corresponding professional fields), it is has moved away from the basic details of programming and into more and more fields of human endeavour, from psychology through sociology into cultural heritage, gaming and art: supporting a huge variety of social experiences [14]. This move can in part be attributed to the fact that in the same period the technology itself has undergone radical transformations. Moore's law not only accounts for a growth in computing power, but there are similar exponential factors in economy, reduced screen size, increased screen resolution, increased power efficiency, and so on. The result of all of these changes is that technology has become more pervasive and ubiquitous, and hence it has reached into and affected more fields of human endeavour.

Alan Turing's famous result that any sufficiently powerful computer (and it does not have to be very powerful) can simulate *any* computer whatsoever has quietly moved from a theoretical insight into everyday life. Turing called it a Universal Computing Machine [30, 31].

When we add high resolution displays, a universal computer can go beyond just simulating anything but can really *look like anything*. Since humans can't tell the difference by looks alone, we really mean *anything*, including non-computable things. We can, for instance, we can watch a realistic computer-generated science fiction animation — where anything that looks like it is working is in fact completely computer-generated. Computer graphics used in popular animated films creates worlds that looks as good as and as realistic as the real world. Moreover, we can go beyond the merely visual: when we add robotic arms, sensors and other everyday features the computer *becomes anything*.

However, this move in both HCI and technology has somewhat left behind "classic" usability problems and yet these are ones that are still important in terms of their impact on users, in some cases being literally a matter of life and death. The user experience is so powerful that less obvious problems, like errors, get less attention. Indeed, errors happen because they are *not* noticed — if we (whether designers, programmers or users) noticed errors, we would correct them. As computers have become more and more attractive, their underlying dependability has not improved as much as we think.

To understand these deep problems, it is necessary to go back to the very foundations of what it means to interact with a computer. When we interact with anything, we, at least implicitly, use a *language*. When we interact with humans we may use English or another natural language, and with programmed computers, whether web sites, PCs, or mobile phones and even door bells, we use much simpler languages. Language is the mechanism to convey meaning, be that a complex expression of our thoughts to a simple single "I'm at your front door."

All computer programs implicitly define a language of either data or interactions (or both) that they can accept and give meaning to. Of course, they do not knowing they are giving meaning to the language but nonetheless they act according to the meaning understood by their developers. If we were building a system to understand English, this would be very obvious — you'd see things like "verb" and "noun" all over the program — but most of the time it is too easy to build systems without thinking clearly about the language and there is nothing visible to show for it. For a door bell, the language of interaction is so obvious that there is probably nothing left visible that specifies or defines the language, or even needs to.

Programmers are taught a lot about languages (regular languages, context free grammars, and so on). And then they promptly forget them because it is possible to program anything without going to the trouble of specifying a language clearly. The move of programming away from low-level details to using packages and complex APIs means that the languages actually being implemented by the computer are well-hidden from the programmer.

It follows that programmers typically build user interfaces *with the wrong languages*, because they never really thought about what the right languages might be. This, of course, creates the entire world of HCI: we need methods and processes to find out what the language should be, and we need ways to help programmers refine the programs to use the right languages, using iterative design and so on.

Design problems are often accompanied by tell-tale phrases that sound like language problems:

"I want to do this, but I don't know what to do or how to **say** it"	…	The system (or the user) has an insufficiently expressive language
"When I **told** it to do something, it did something else"	…	the system had a different semantics from the user

These problems often arise because the programmer has specified a simpler language than the user needs or expects. Part of the problem is that the right language is not obvious. On the other hand, if programmers carefully worked out the right language, they'd spend so much time planning and working with users they'd never get round to writing any useful programs — we are not blaming programmers, we are pointing out that implementing the wrong language is almost always inevitable because life is too short.

If programmers implement a simpler language something very interesting happens. The language they implement is easier for the computer (and for them to program) but probably at the expense of being harder for the human user. The computer program does do something, but it is not as powerful as users might wish. This is the what HCI is all about fixing. Yet because this problem is invisible, it remains a fundamental problem for HCI.

This chapter puts a name to the problem: **premature semantics**. Of course HCI tries to solve the problems caused by premature semantics, but it does so by "fixes" — processes, such as iterative design, that come too late to avoid the *premature* part of the premature semantics. If we can recogniser and know what premature semantics means, we can take steps to avoid it, and hence vastly improve normal HCI methods.

We will build up to our discussion of premature semantics by reviewing how user interfaces and languages intertwine — using regular languages and other fundamental computer science concepts — and then premature semantics will fall out of our discussion naturally. We will show that the idea exposes very clearly some serious issues in user interface design that have been widely overlooked, and certainly not attributed to a common factor.

3 Motivating examples

In order to make our ideas concrete and meaningful to the reader, we will ground many of our examples in a very familiar user interface that has been around for more than fifty years, the calculator. We had thought of using a new user interface like WhatsApp (www.whatsapp.com) to illustrate our ideas about HCI, but interesting user interfaces tend to be complicated and any such choice for our example would leave some readers unfamiliar with critical details. Moreover, WhatsApp is fun, and that distracts from some of the issues we want to examine. So instead, we will use familiar calculators.

Calculators are a very simple technology we have had since before the start of HCI as a recognisable discipline, but which, despite briefly gathering some attention from HCI researchers [18, 33],

they no longer attract the field's attention. A cursory consideration might decide that they are too trivial to consider deeply — "this is how they are." Their persistence is a quirk of fate or history, like the QWERTY keyboard, and research in improving is fraught with pointless difficulties. However, on the contrary, we show that there are subtle problems in their design. This is because of the dominance of **premature semantics** in their interfaces. This problem is not confined to calculators but can be seen in online forms and secure interactions like banking, and more. Our analysis also shows that solving the problems of premature semantics are neither a simple matter of task redesign nor of better programming but requires substantial new approaches to thinking about interaction — and requires new approaches that have yet to be developed.

Calculators were one of the first digital devices to have widespread ownership. Mechanical digital calculators gave way to electronic and then to computer-based calculators very rapidly. Almost every computer — mobile phone, desktop PC — provides a simulation of a 1970s style calculator because we are so familiar with them. The basic design idea has hardly changed despite the radical changes that have occurred in computing over the years since their first introduction.

Calculators are simple, well-understood devices. We understand what they are supposed to do. Probably, we were all taught about them at school, and they haven't changed much. There is therefore no excuse for them to have poor user interfaces.

Yet in 2000, Thimbleby noted that despite decades of use, development and research [33], calculators still had horrible usability [18]. The problems persist to today [19].

3.1   Calculators are *still* needlessly bad

Being charitable, we might say that some of the problems identified were originally due to the physical construction of early calculators and then persisted because of their position as cheap, disposable consumer electronics. Thus, there were originally many physical constraints on the design of such devices and not enough profit to make substantial redesigns worthwhile. Of course, there has been decades since then to improve and, moreover, the "mobile revolution" has occurred over the intervening time. Almost all smartphones have a calculator app so all of the physical constraints of older calculators have been removed and the investment in redesign is made in the move to software in any case. Yet remarkably, these entirely digital calculators are no better. And in fact, as we now demonstrate, they have unexpected problems. Something very strange is going on. It is even stranger that nobody else is noticing, or worried about the consequences of poor systems.

We know users make mistakes, and almost all user interfaces provide features so that users can correct their mistakes. The delete key is the simplest example. In a word processor, web form, or even devices like car parking meters, the user types text (like their car registration number). If they make a mistake, they hit the delete key, and the last thing they typed disappears, so their error is corrected and they can carry on and try harder to do what they originally wanted. Let's say I want to type "their" but I start to type "thi. . . " — I missed out the e or typed the i too soon. I hit Del (delete) and the text turns into "th" and I can then carry on and type "eir" with the final desired end-result of correctly typing "their." The delete key works when I type anything; you would not expect it to behave differently if the example was about correcting errors in typing a car registration number for instance rather than English words.

But now consider typing a number. If the delete key works the same way, when you make a mistake in typing a number, you can delete errors and hence correct them.

Strangely on almost every calculator, the delete key (when there is one) works in very strange ways.

1. On the Hewlett-Packard EasyCalc calculator, the delete key does not delete decimal points but only digits.

2. On the Apple iPhone calculator, the user interface does not allow numbers with more than one decimal point, so if you delete a second decimal point you keyed by mistake, doing this

deletes the only decimal point, leaving you with no decimal points rather than one as you expected.

3. Many calculators have a limit on the number of digits they can display. For concreteness, suppose a calculator can display at most 8 digits. If you key 9 digits and delete the last accidentally excess digit, you will end up with 7 digits, not 8 as you expected.

We have found many people say, "Well that's what they do!" But just because we are familiar with a problem does not mean it is harmless and that we should not try to solve it. Using computers and other complex devices may well be a form of hazing, which initiates users into the fraternity of experts; and once hazed, you disdain the naïvety of people who see problems rather than commit to learning the rituals that would save them and have saved you. (We will consider a more constructive way of talking about hazing in section 6.4; more constructive in that it suggests some solutions.)

3.2 Entering long numbers

Obviously, there is no limit to the size of numbers in principle and so there is no limit on how many digits a person may wish to enter into a calculator. This of course causes a design problem for how many digits a calculator can display. Some calculators, following the tradition of the early calculators, simply prevent making numerals longer than the display of the device. So for instance, the Windows 10 calculator allows a user to enter 16 digits (in standard mode) after which no further keypresses are accepted.

However, the Apple iPhone calculator simply makes the font smaller and smaller to fit more in the display area of the calculator, and then at some point it starts to discard the user's digits without warning. On Apple macs, the calculator makes the font smaller and smaller, so that it eventually becomes unreadable at about 1mm high, and then it starts discarding the most significant digits. These tricks show the programmers have made choices, premature choices, about the problem, but they have not made the calculator easier or more reliable to use. It seems strange: clearly the programmers thought about the problem, took steps to mitigate it, but yet failed to do it adequately.

There are boundary cases where, as the user approaches the limit of the display, the representation of the number becomes unstable. For example, on the iOS v7.1.2, the calculator that comes as standard on the iPhone allows users to enter up to 16 digits in scientific mode (which appears when you hold the phone on its side, in landscape). So if you enter fifteen 9s there is of course one last digit that can still be entered before the display is full. And this last, sixteenth, digit leads to very odd behaviour. If it is a 9, the display changes to exponent form and displays `1e+16`. Moreover, deleting the last digit results in 1 followed by 15 zeroes, which is not what you might expect from `DEL`. If the sixteenth keypress is a 3, the display becomes fifteen 9s followed by a 2. Similarly, entering a sixteenth digit of 5, 6, or 7 all results in the last digit in the display to being a 6.

Arguably, at such highly precise levels, a unit difference at the sixteenth significant digit is not often going to matter. But who knows? We certainly do not, but we think it is highly likely that the programmers of the calculator do not either. And the way `DEL` works is totally unlike the way it works on everything else. What this example implies is that the process of entering a number in this calculator is not simply one of entering a string of digits but is in some unfathomable way tied to meaning of the number (or something) as it is internally represented in the calculator. Even a six-year old child understands that the current behaviour of any calculator is peculiar (Patrick Cairns, personal communication).

These examples come from looking at specific boundaries prompted by our current concerns for number entry. It is impossible to know what other boundary cases arise in the multitude of other functions that calculators possess. What is very clear is that calculators are not simple user interfaces. Indeed, the way the way they are designed to handle errors (e.g., using delete keys) can make errors worse.

3.3 Why calculators should be better

These problems illustrate that even the conceptually very simple task of entering a number into a device is not without usability problems. In some ways, this task is almost so small as to make it not worth considering because obviously it is easy to re-enter a number into a device when you make a mistake. But there are four responses to this.

1. The problems may be small but they are hugely widespread. Problems like these occur not only in the various designs we mentioned but in the design of all calculators that we have examined. Such ubiquitous problems may have small individual cost but have huge cumulative cost, that is, when multiplied up by the numbers of users of calculators and the number of times they are used by each user. For instance, nurses use calculators every day to work out drug doses for patients: one wonders how many design-induced errors happen with catastrophic results.

2. There is a secondary cost to this ubiquity in that we have to take time to teach our children what the problems are instead of solving them. Schools teach children not to trust calculators and to try performing calculations in various ways. It is like teaching people how to drive a car that doesn't have very good brakes instead of working out how to make safer brakes.

3. Calculators are only the most familiar representative of the type device that requires number entry; there are many variations on the number entry task in many other devices and we have found that these too have similar or related problems [2, 19]. And when the devices considered are aircraft altimeters or medical infusion pumps, these fundamental problems become extremely important. People often die in hospitals from "out by ten" errors where a drug dose is ten times (or more) too high. We wonder how often this happens because correcting a number entered like `1..5` to be `1.5` gets mangled *by the bad design* to become 15. Nobody knows.

4. We have been using Arabic numerals in the West since Fibonacci published his best-seller *Liber Abaci* in 1202. Although there were a few issues sorting out decimals and negative numbers since then, by the twentieth century, certainly, Arabic numerals were perfectly well understood and, surely, there is now no excuse to program number entry systems improperly.

Something very interesting must be going on.

4   Finite state machines and HCI

Whenever a user interacts with a system, there is the expectation that the system responds in some way; that is, on each user interaction, the system will normally change. To be precise, we can say it should change **state**.

In some cases, the change of state will be quite innocuous. For instance, pressing `1` on a calculator can make a `1` appear in the display. In some cases, the change of state is quite important such as pressing `=` on a calculator will lead to the display being cleared and the result of some calculation being presented, as well as getting the calculator ready to receive new numbers for further operations: the change of state here is quite complex. In some cases, a change of state may be very important, such as changing a setting on an autopilot in a commercial airliner can make the difference between setting a target altitude or setting the rate of descent, which can have very different end states [4].

From the earliest days of computer science, it has been recognised that many algorithms and interactions can be understood as the system moving through a serious of different states and this

led to consideration of an important abstract idea called Finite State Machines (FSMs) or sometimes Finite State Automata (FSAs).

FSMs are very well established theoretical tools in computer science, though not always considered as practical tools. They have also been used in HCI to model interactions and to allow various sorts of analysis.

We can explain FSMs in many essentially equivalent ways:

- Pictorially, each state of a FSM can be drawn as a circle, and each transition from one state to the next as an arrow joining two circles. Typically the circles will be labeled with the states or modes they represent (like "on" or "off") and the arrows will be labeled with the names of the buttons or other actions that cause the changes of state. We'd expect, for instance, an off arrow to point to the off state circle. Note that there may be lots of arrows for the same action — many states can be switched off by pressing $\boxed{\text{Off}}$ so there may be lots of off arrows, one from each state (at least from those that can be switched off). However, there is only one circle for each state; there is usually only one Off circle, for instance. (There may be several off states if the device remembers things when it is off; if so, it needs different states to remember exactly how it is off.)

  See figure 2 for an example. As shown in figure 2, normally one state will be marked as the starting state — typically the state when the device is off.

- Mathematically, a FSM can be understood as an abstraction of this pictorial representation in the form of an abstract graph, where circles correspond to elements of a set of states, called nodes, and arrows ordered pairs of nodes called edges that are mapped to labels via a labelling function. Graph theory is a substantial part of mathematics, though confusingly "graph" is also used as the term for diagrams to plot functions like $y = x^2$.

- In software, an FSM can easily be represented by numbering each state, numbering each user action (e.g., each button), then using an array of state numbers T, to represent the FSM. When the user does action b in state s, the state is simply updated by doing the assignment s = T[b,s].

- Using algebra we can describe a FSM very abstractly. There is a set of states S and a set of transitions (typically user actions, button presses, hand waving, etc) T. Each transition t is a function that changes the current state to the next state: $t \in T\colon S \to S$. A FSM starts in an initial state $s_0 \in S$ and is operated on by a sequence of transitions $t_1; t_2; t_3 \ldots; t_n$ and it will then be in state $t_n(\ldots t_3(t_2(t_1(s_0)))\ldots)$. Typically a special transition "reset" (such as removing the batteries or unplugging it, or pushing in a paperclip into a special hole) will put the FSM bak into the initial state s_0.

- Because FSMs are such basic yet ubiquitous things, there are many other alternative ways to represent and understand them. For example, each possible user action in a FSM can be represented as a single matrix, and then the effect of a sequence of user actions can be represented by the matrix product of each action matrix [20]. This is a remarkable result as it proves that user interfaces and matrix algebra are essentially "the same thing." Since mathematicians know an awful lot about algebra, they also know an awful lot about interactive systems!

It is an important fact that every computer system can be described as a FSM. Possibly some trivial additions may be required — e.g., a lottery computer will need to have some randomising transitions to generate lottery ticket numbers — but *every* computer's user interface can be understood as state transitions caused by user actions. As *any* program represents a FSM, often the programmer will program "how they like" and the FSM is then likely to be very hard to see — but a FSM is still there implicitly and can be "discovered" by special techniques [26].

Despite their power and generality and their rigorous mathematical basis, FSMs have got a poor reputation.

We can easily draw a few circles and arrows and thus understand how FSMs work (they are no harder than the children's game of snakes and ladders), but then it is tempting to think that FSMs *have* to be simple enough to draw. If so, that would limit FSMs to a few states only — any more and they would far get too tedious to draw. In fact, FSMs work the same way even if they are so large we cannot sensibly draw them. FSMs need to work on their public image!

An analogy may help. We can "draw" the number 6 by drawing ⠿, and this familiar way of drawing numbers makes numbers very easy to understand. Adding one to six is easy to draw, and when we were children drawings like this helped us learn how numbers work. But we can't usefully draw 1,007 this way — it will look much like any other large number, like 999 for instance. Obviously, we can only sensibly draw relatively small numbers, but nevertheless we know that large numbers are still very useful even though we cannot draw them reliably.

Note that we can add 6 and 2 by simply putting our drawing of 6 next to a drawing of 2:

$$\text{⠿} + \text{⠃} = \text{⣿}$$

We get 8 with no effort! In other words, we can add up *without consciously adding*. Indeed, if we *delete* the + symbol, the numbers "just" add themselves.

Similarly, all programs are using FSMs, even if the FSMs cannot easily be drawn or visualised. A bit of program like `if( x < 0 ) x = -x` is doing something an FSM can do, but you cannot see the FSM in the code — just as in ⣿ you can't see the + sign.[1]

Another problem FSMs have is in the "programmer's imagination," which is that programmers think FSMs are technically limited compared to Turing Machines and general programming. This is true in infinite theory, but not in practice. Any real PC is a finite state machine — it is not an infinite machine like a Turing Machine. The practical truth is that FSMs do not have good ways to abstract them, so programmers tend to write in Java or some other convenient programming language. Their programs are equivalent to (big) FSMs, but programmers "think in" Java, not in FSMs.

4.1   An example FSM

Figure 1 shows a UK Coast Guard phone near the sea. Imagine watching somebody getting into difficulty in the water: you run to this emergency phone to call the Coast Guard.

As you run towards it, the big writing on the sign primes you to press 999 (the UK emergency number, as opposed to 911, 112, etc, which are used elsewhere). But when you get closer and can see the details of the user interface, just *how* are you supposed to do that? One thing we can be certain of, is that priming you to do something impossible is going to make your emergency call harder.

This is a serious design issue that could have been prevented if the user interface had been represented as a finite state machine. As is clear from figure 2 there is a transition between two states that has *three* buttons that are used nowhere else in the design — something very strange is going on in the design. In addition, giving the user unnecessary choice slows them down (Hick's Law), here further delaying calls to the Coast Guard — it'd be better to label all the buttons 999 anyway! (Notice the button labels shown in figure 1 are blank and could have been filled in with 999 — or, better, each button could have been labelled 999 instead of 1, 2 or 3.)

It would be easy to write a program to automatically find such errors, and many others, in the graphs of designs before they are built, but unfortunately here it is too late for the the Coast Guard to fix the problem easily.

---

[1] One of us has a car whose user manual has the warning that its clever computer features do not help the driver overcome the laws of physics. In other words, the car obeys the laws of physics even though the driver cannot see them, and — as the manual tries to warn — even if the driver foolishly denies the laws are there anyway. Similarly, the laws of FSMs are obeyed even if they cannot be drawn or seen.

Figure 1: A very simple walk-up-and-use user interface. A UK Coast Guard phone (photographed in 2016) for the public to use to call the Coast Guard in the event of an emergency. In the UK, the standard emergency number is 999 — as the sign makes very clear. But the phone itself does not have any way to dial 999, so what should you do? It is not very clear! Presumably only three buttons (labelled 1, 2, 3) were provided either to make it "simpler" or to save money? Figure 2 shows the underlying FSM for this phone, and the FSM shows how to use the phone to correctly call the Coast Guard.

4.2   Using FSMs for interaction

It is easy to see how FSMs can represent many sorts of devices. Push button devices, such as DVD machines and calculators, move between states as a consequence of the user pressing buttons, the buttons provide the labels for the edges (arrows, state transitions). Simple websites display different pages, each page representing a different state, and move between pages/states as a consequence of a user clicking on links. Web sites can simulate any interactive device, so obviously since a web site is a FSM, then FSMs can simulate any device. It is a routine problem to use an FSM to see whether it is possible to navigate from any webpage in a website to any other, so it follows it is a routine problem to use an FSM to analyse accessibility for any interactive system. In fact, FSMs can be used with computer tools to analyse many interaction properties very easily — recall that a FSM is easy to represent in a program as an array, and we can automatically check that array for a large range of interaction properties [6, 21].

Here are some typical questions that can be directly answered by analysing FSMs:

1. Can the user reach certain states?

2. Can a user always undo any action?

3. Is it possible to get into any state from anywhere?

4. Are there states that dispense chocolates follow states where no coins have been inserted?

5. Can the device be switched off by pressing OFF in any state?

6. If a burglar presses digits at random, how long do they take before guessing the code?

Even from thinking about simple examples like an oven temperature setting, it is clear that what state the system is in has a significant impact on what actions are possible or safe. Where a state is
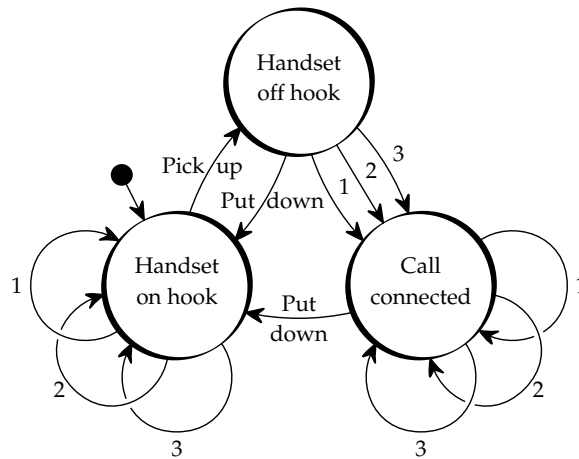
9

Figure 2: A drawing of the finite state machine for the Coast Guard phone shown in figure 1. The phone starts in the state indicated by the ⬤↘ arrow (pointing to the bottom left circle, "handset on hook"), then user actions follow arrows to the other two states. For example, in the start state, pressing any of the digit buttons (1, 2, 3) does not change state, so nothing happens — in this state the buttons do nothing. The user has to pick up the phone handset to start anything happening.

not otherwise visible, for instance the oven is above $40°$C, it is important to indicate this to the user some other way such as a warning light or door lock. Again, we can automatically analyse FSMs to ensure they have such important properties.

Asaf Degani [4] uses state machines to consider the interaction with autopilots in airplanes. He shows how interactions with the autopilot can lead to unnoticed state transitions, particularly when the environment is providing inputs to the device such as the plane coming close to its target altitude. Where such transitions are not made visible to the pilots, it is impossible for the pilots to know what state the autopilot is in and therefore reliably predict the consequences of their interactions with it. Degani shows how this problem of hidden state transitions has led to fatal and expensive accidents.

With FSMs, it is also possible to add information to the transitions. For instance, if numbers between 0 and 1 are added to the edges to correspond to the probabilities of a particular button being pressed, then is possible to calculate the probabilities of a system ending up in any particular state (in fact, we end up with a Markov model). We can then estimate how many steps it takes to get to any states of interest. We have previously used this idea to show that an error, say in setting a microwave to heat something up for a fixed time, can lead to users being unable to set the microwave at all. But, also, the introduction of a [Quit] key, and the knowledge to use it, can greatly help people from getting trapped when that happens [25].

It can be seen that FSMs make a bridge between the theoretical tools of computer science and the ability to analyse interactive systems. It is probably also clear that effective analysis of real systems requires support from suitable tools that can manage the generally large sizes of real FSMs. Some such tools, like MAUI and others [6, 21] exist and can help. However, analysing FSM models is one thing and building working programs is another. What is still needed is a way to move fluidly from the analytical power of FSMs and the actual systems being developed so that a programmer can do the job of programming but also exploit the analytical strengths of FSM for interaction design.

Unfortunately because programmers rarely use explicit FSMs (e.g., as simple as the array we illustrated above), even these simple questions are usually very difficult to answer. In fact, they are so hard to answer they are rarely even asked. It is arguable that major advances in HCI, such as the initial explorations of GOMS and KLM [3], floundered because what an HCI or UX professional is interested in is just too hard for a programmer to address. Thus GOMS was used for really very

trivial systems, and still hasn't been used for anything of ordinary complexity.

4.3   Regular expressions

Another long-established tool in computer science is the regular expression. Regular expressions are a way to define formal languages — artificial languages with a clearly specified grammar. Regular expressions tend to be used with reference to problems like whether a given string of characters is a valid word in a particular language.

For example, YN16 AGY is a valid UK number plate for a new car, but YYYYX1 is not (for simplicity we will ignore the possibility of historical numbers that predate the current rules and "vanity" numbers, like EGO 1, which can be bought at a price). Regular expressions can easily be used to specify the words (i.e., registration numbers) of the registration plate language.

A regular expression to capture the format of valid number plate registration numbers is:

```
L = 'A'|'B'|'C'|...|'Z'     L can be any letter, A or B or C . . .
D = '0'|'1'|...|'9'         D can be any digit, 0 or 1 or 2 . . .
Pl = LLDD' 'LLL             A valid registration is letter, letter, digit, digit, space . . .
```

There are many other notations for regular expressions that can be used to say the same thing. Many readers will be more familiar with the Unix notation, which in this case would be

$$[A-Z][A-Z][0-9][0-9]\quad [A-Z][A-Z][A-Z]$$

and even then this is only one way Unix can express registration numbers. Our own notation used above has the advantage that we can *name* regular expressions — such as using L to represent letters — and then use the name L later to save rewriting the same idea multiple times (making it much less likely we make mistakes). For example, if we sorted out how to handle that the UK registration plate font makes the letters/digits I and 1, O and 0 identical, we only have to fix this in two places (in the definitions of L and D), rather than in seven where letters and digits are needed. Conversely, if there is a bug in our regular expression, then the same bug will reappear up to seven times: this might sound like a disadvantage, but we are also more likely to spot bugs in the first place, which is a huge advantage, and when we fix any bug we find, we actually fix seven. This is a very powerful advantage.

For real UK registration numbers there are actually more rules than this. For example, the pair of digits in the middle of the number plate represents the year of manufacture of the car, so for instance anything above 17 but below 50 is not currently valid (at the time of writing) but we did not specify this above. Ignoring such details, then, more or less Pl defines valid number plates and would, for instance, exclude many number plates from outside of the UK. Thus, regular expressions can be used to both define valid things and to check whether given things are valid or not.

The three basic operations permissible in regular expression are sequence, choice and iteration.

Sequence is the operation of being given two regular expression, you can have one expression follow on from another. This is precisely what is seen in defining Pl; a valid registration is a letter *followed by* a letter *followed by* a digit . . . In most regular expression notations, there is no special symbol for sequence: thus, "LD" means L followed by D.

Choice is that a new regular expression can be made out of two existing ones; it is either one or the other of them, that is for regular expression A and B, C= A|B means C is either A or B. Here the symbol "|" is the notation for choice. The expression L above is a short hand of listing the choice options that L is one of the characters of the upper case roman alphabet. It could be expressed as L = 'A'|'B'|'C'|...|'Z'.[2]

---

[2]This example makes it look like choice allows 26 choices — more than two choices! In fact, if we write the choices out explicitly each with exactly 2 choices: (('A'|'B')|'C')|...|'Z' or as 'A'|('B'|('C'|...|'Z')), etc, these regular expressions are all the same, so the brackets are unnecessary. In other words, choice is associative.

```
Account number:
<input
   type="text"
   autocomplete="off"
   id="pnlSetupNewPayeePayaPerson:frmPayaPerson:stringBenAccountNumber"
   name="pnlSetupNewPayeePayaPerson:frmPayaPerson:stringBenAccountNumber"
   maxlength="8"
/>
```

Figure 3: HTML for a user to enter an account number, copied from a Lloyds Bank web site in 2016. Notice the code `maxlength="8"` (on the penultimate line) which will *silently* truncate any account number to a maximum of 8 digits: neither the user nor the bank's server will know.

Finally, a given expression can be iterated (repeated) a number of times to form a new expression, represented as `B = A`* which means `B` is any number of repetitions of `A` including no repetitions. If a $+$ is used instead of a $*$ then this means `B` is at least one repetition of `A`. Sometimes it is useful to be specific, and write `A`^{n} for repeating n times or `A`$^{m-n}$ for repeating between m and n times. We did not use these ideas above because car registration numbers are too simple to need them — writing `Pl = L`^{2}`D`^{2}`' 'L`^{3} is more confusing that `Pl = LLDD' 'LLL` which is what it means. In more complicated cases, though, the additional notation can be very useful (e.g., it is much easier to tell the difference between `D`^{8} and `D`^{9} than `DDDDDDDD` and `DDDDDDDDD`).

Regular expressions are the tool behind many familiar user interactions like making sure users enter a date in the correct format or enter a number when they should. However, even in these simple interactions where regular expressions are useful (and would help ensure user interfaces did what they were supposed to), they are often not applied when they might be.

Like FSMs, regular expressions also have a public image problem. They are fantastic for performing string operations, like finding car registration numbers in a big file of text. Many programming languages (e.g., JavaScript, PHP, etc) provide powerful built-in features for using regular expressions . . . for string operations. But just because they are great at string operations does not mean that is all they can do, and in particular that they should not be used right across all aspects of user interface design.

Greta Fossbakk lost 50,000 Norwegian Krone (about US$60,000) due to a poorly programmed user interface [12]. Fossbakk entered an account number but accidentally pressed one digit twice. The resulting account number was of course then too long, but the bank's system truncated it, and the truncated number happened to be a valid account number but unfortunately the right number for another person. Fossbakk did not notice this error, and confirmed her money transfer — to the wrong account. Of course the bank argued Fossbakk had confirmed what she wanted to do, but she had confirmed a simple error that the bank should have — and could have — detected.

A regular expression in this case could have been used to validate Fossbakk's entry, and would have detected her error because it was too long. For example, `D`^{8} would have done. The point is, had the programmer specified the user interface (here, using a regular expression), the problems would have been obvious, and a solution found (e.g., warning the user). In fact, using regular expressions to validate user input is an obvious professional decision. In this case, though, it seems the programmers did not specify anything, but rather "it just happened" and then *nobody thought about it*.

The user interface (if designed properly) should have forced her or the bank to re-check the input more carefully. This may seem like a rare and unlikely feature of one particular bank but it is not. Figure 3 shows some code taken from the UK Lloyds Bank's current (2016) web pages for the same task. Lloyds Bank uses HTML to truncate account numbers to 8 digits, and it does so silently. Because the web browser (thanks to the HTML) truncates the number, the bank's servers have no idea that an invalid number has been entered.

Though we rarely think about it, Arabic decimal numbers (and indeed all number systems if

12

they are reliable) have a well-defined structure that can be captured by a regular expression:

```
1   NZ = '1'|...|'9'          any non-zero digit
2   D = '0'|NZ                any digit, including zero
3   Pt = '.'                  decimal point
4   Int = '0'|(NZ D*)         whole number part
5   Fr = D* NZ                fraction part
6   Num = Int|(Int Pt Fr)     a number is either an integer or a fractional number
```

These rules may seem unduly complicated (and we did not include any rules for signs, + and −) but they are not really complicated:

The final rule 6 is saying that a number is either a whole number (an integer) or an integer followed by a decimal fraction. Rule 4 is saying that a valid integer does not start with 0 unless it is actually 0, and rule 5 says that the decimal fraction of a number does not end in a 0. Actually, there are contexts where that is valid, for instance to show the number of significant figures but in general, if we are representing a precise value then rule 5 is necessary.

These rules make `Num` specify whole numbers (which do not have a decimal point) and fractional numbers (which do have a decimal point). We have not permitted a number to start with redundant zeros, and we have not allowed a fractional number to end with a zero: thus `00050` is not allowed to represent 50, and `0.50` is not allowed because of its trailing zero. Moreover, we do not allow "naked decimal points" — so neither `.5` and `5.` are allowed. The naked decimal point rule is inspired by the Institute for Safe Medication Practices (ismp.org), which forbids naked decimals as they encourage errors — for instance, a nurse could misread `.5` as `5` and give somebody a drug dose ten times too high.

When it comes to a task like number entry, it seems unlikely that checking the number is actually a valid number would make much difference to anyone — and few user interfaces check anyway. However, we found that in the possibility that users make errors when entering numbers, then simply checking if the result was a valid number reduced the risk of high impact errors by a half (where by high impact we mean when the number, perhaps a drug dose, is out by a factor of 10 from the correct value) [24]. Unfortunately many programmers learn about regular expressions as a theoretical computer science curiosity and never realise they are very useful in user interface design!

4.4   Regular expressions = FSMs

Though at first sight, regular expressions and FSMs look very different, in fact they are closely related.[3] Any FSM can be expressed as a regular expression and any regular expression can be transformed into a FSM [1].[4] The basic idea is that if a regular expression defines which sets of characters define valid words or sentences, the FSM is able to check whether one is actually valid. It does this by starting in a specified state and then matching the letters of the word to labels on its edges. If, when it reaches the end of the word, it is in a special final state then the word is valid. If not, then the word is invalid. Thus, what looks like a complicated checking procedure of matching words to regular expressions becomes an entirely algorithmic process:just running a FSM. In this sense, regular expressions are a more humanly meaningful way to represent FSMs. A FSM may be hard to draw but its regular expression is usually much easier to write.

In fact, the operations of regular expressions (sequence, choice and iteration, described above) are *exactly* what FSMs do by using transitions. Thus, for instance, the regular expression choice `'x'|'y'` does just what a FSM in some state does when it makes one transition on the action x and another transition on y.

---

[3]Technically, a FSM can end up in any state, but a regular expression can only end up either succeeding or failing to match its input; hence regular expressions are exactly equivalent to special FSMs called finite state acceptors, which are FSMs with exactly two designated final states. Such final states can always be added to any FSM to make it into an acceptor.

[4]There are some easily addressed technical differences between FSMs (and non-deterministic FSMs) and REs but it is beyond the scope of this chapter to worry about it further.

4.5 Programming with FSMs

Programming is complicated and is an activity that induces many well-known cognitive problems. We have brains of finite capacity, and the more work they do on a task it is inevitable that they have less spare capacity to work on other tasks or even consider all the peripheral issues. This is problem is called loss of situational awareness or tunnel vision. Put in other words: getting a program to work at all is so hard that making it easy to use is likely to slip to second place, and, moreover, the programmer will be unaware this is happening — their brain is full. To be aware requires cognitive resources: loss of situational awareness by definition is unconscious.

One way to help is to use teamwork (so other people help you look out for things you are missing), but not the sort of teamwork that Landauer [9] envisages of one team member (the UCD expert) telling the programmer what to do — which just makes the programmer's job harder — but means being present with them and helping them do it better.

Another way to help is to use programming techniques to help manage the complexity of programming. Software engineering is the discipline that studies this, and programmers should be trained in its techniques [17]. One of the key techniques is **separation of concerns**, to split a programming problem into separate components with as little interaction between them as possible. Then the separate concerns can be addressed independently. An example of successful separation of concerns happens on most web pages: the content, the activity, and the presentation are managed by separate parts of the site — in HTML, JavaScript and in CSS (to name a few). Certainly the boundaries are often blurred, but in principle the areas of covern can be separated, and systems become "cleaner" and easier to develop and manage the sharper the separation — they also become easier to update in response to UCD requests. Note that these points apply to any interactive system, not just to web sites; indeed, web sites are a good way of prototyping any interactive system, and hence exploring the best separation of concerns.

A FSM provides an excellent separation of concerns, as it separates the "machine" delivering the style of behaviour the user requires from the application and presentation details. For example, clicking buttons needs implementing (for example, so that a screen image changes as the mouse moves over it and when the mouse clicks on its active region), but what clicking a button does for the user is a different problem. As a useful side-effect, this approach makes the user interface more consistent for the user:

1. How do buttons work? What are the timing issues for double clicks? What is a long hold? Do buttons audibly click when pressed? What happens when the user clicks on a button and moves out of its active region before releasing the mouse?

2. What should the application do when a user clicks on a button? What are its modes, and how do buttons change their meanings in each mode?

3. How does the application achieve what it should do?

Step 2 above can often be done by a FSM, and doing so creates a useful wall — a separation of concerns — between steps 1 and 3, which makes those steps easier to implement reliably and consistently. Otherwise, there would be a temptation to implement particular buttons that do particular things one at a time, which would entwine the button and what it does, and hence each button would be implemented in a slightly different way.

A corollary of separation is that program code gets reused. Instead of every button being implemented in its own way all the buttons can share the exact same code. This means that any bugs in button design become apparent quickly — because testing any button is the same as testing all of them. In contrast, when buttons are implemented one at a time, then each button must be rigorously tested, and that is hard work. Separation of concerns concentrates design effort into a few important places, and those places get greater scrutiny than without the separation.

Not only does it make it easier for the programmer (and hence ultimately easier for the user because the program is better implemented) but the separation of concerns means the *meaning* of

the interaction has been separated out and becomes a thing-in-itself. Once a FSM is used, it can be analysed.

For example, whatever a user interface looks like (step 1 above) and whatever it is doing (step 2 above) we probably do not want the user to get stuck. There are many ways of expressing this design requirement, but suppose it is expressed as "whatever the user is doing, they always have the option to do anything else." If this is the design requirement, it translates to a *trivial* test on the FSM — is it strongly connected? Another form of the requirement (with slightly different connotations) is "the user can always undo whatever they have done." If that is the design requirement, again it translates into a simple test on the FSM: is it symmetric?

Often, strict requirements like these simple examples are problematic or raise further design questions that need carefully exploring. For example, once you start using a fire extinguisher, you cannot go back to any state of not having used it — so fire extinguishers fail both of the properties above, but they are no less useful for failing them. On the other hand, activating a fire alarm, you might want to undo because it may be an accidental activation, and setting off a fire alarm is expensive (e.g., all staff have to stop work and leave the building).

Once a FSM is used: critical user interface design questions can be asked, questions can be answered, and the design trade offs are very easy to explore. These are advantages for programmers, designers and users that conventional programming rarely benefits from.

4.6 Going beyond FSMs

FSMs are very good for designing simple systems (ones we can draw or express as regular expressions), but as we've emphasised they can become complex too quickly. Put more positively: FSMs therefore help keep designs simple because the effort of designing complicated ones hopefully dissuade designers from unnecessary complexity. Unfortunately, it's far more likely UI designers will just start developing in completely *ad hoc* ways without using anything rigorous like FSMs. Programmers will be unlikely to correctly implement that which has been vaguely defined.

For the purposes of this chapter, though, FSMs highlight in a very clear way many user interface programming issues and trade-offs. FSMs are a type of *model*, in fact a very elegant and clean type of model, and their trade-offs are typical of trade-offs with any modeling approach.

Perhaps the most important and well-known bridge between the simplicity of FSMs and the intricacies of real life applications is StateCharts. StateCharts are a concise way of drawing interactive systems that designers can easily use, but they allow *much* more complex designs that can easily be handled with FSMs [21]. Yet StateCharts do not lose any of the technical advantages of FSMs. Furthermore, StateCharts have many design and analysis tools, such as those available to support UML (StateCharts are part of UML).

Unfortunately we do not have space to discuss StateCharts or UML further here — in some way, it would be like us telling you to use Java or C++ to program. It's an interesting choice, but the necessary discussion to understand the choices will divert us too much from user interface design and premature semantics.

There are others forms of model based programming for user interfaces, and their advantages and disadvantages are very similar. All help programmers implement better programs and to reason about them. We recommend Jackson's *Software Abstractions* [8] as an introduction to substantial bigger field.

5 Premature semantics

Careful consideration of calculators makes it clear that a calculator, whether a mechanical device, electronic or entirely computerised, always displays a valid number in its display. That is, the semantics of the input are forced to be a number *all the time*. For example, if you try to enter two

decimal points in a calculator entry, the second one leads to no change because a valid number cannot have two decimal points. More generally, we call this issue **premature semantics**: the semantics of an input are fixed (to be a valid number) before the entry is finished. Note that the premature semantics affects the user — input is a number before they have finished (they accidentally entered a non-number, but something was mangled to force it to be a number). That means the user's ability to correct their own error is compromised; for example, the ⌈DELETE⌉ key won't work as they expect.

The reasons why calculators are like this are easy to imagine. Early electronic calculators would hold the input number in a register inside the calculator. The device could block any input that would lead to an incorrect number or to overflowing the size of the register. And, because the physical devices have a limited number of buttons, this was easy. In addition, the early devices did not allow delete, as we would understand it now, but only a ⌈Clear⌉ function which cleared the value in the register rather than remove individual digits.

Though they were almost certainly not programmed this way, it is straightforward to understand the interaction of entering a number in an early calculator through a FSM. With the underlying principle that the calculator always displays a valid number, there is (or there is in principle) an underlying FSM that accepts keypresses that lead to valid numbers and blocks keypresses that do not. So the display starts by showing 0 (a valid number even though the user has not even started entering a number) and as the user presses keys, the digits are accepted and appended to the display. A single keypress of a decimal point is a valid state transition in the FSM so the point is appended to the display. Having entered states with a decimal point, any further decimal point keypresses do not define valid state transitions so they are blocked. Pressing ⌈Clear⌉ resets the display and the FSM back to the initial state.

It is worth contrasting this with text entry. Because valid words in, say, English do not have a rigid structure that can be easily captured with a regular expression (or arguably any formal expression), it is not so tempting to enforce premature semantics in text entry. A spell checker can only identify an error once a word has been entered and even then, as we commonly experience, a correctly spelled word is not necessarily semantically rite. Sentence level checks on semantics are even more difficult. Certainly, there is no way to enforce semantics in text entry as the text is being entered.

Some systems do impose premature semantics on text. As you write the space after, say, "thankfull" it magically turns into "thankful." But if you are copy typing and not watching the screen, you would type "thankfull ⌈DEL⌉⌈DEL⌉ " and you'd end up with "thankfu" — since both you and the system deleted the extra "l."

We now consider how premature semantics leads to two specific types of problem: problems of construction and problems of representation.

5.1 Problems of construction

Whenever a person enters information, be it digits, text, photographs or whatever else, into a system, there needs to be a process for the user to construct the input. But of course, people make mistakes and want to correct mistakes and for this reason, keys like ⌈Clear⌉, ⌈C⌉, ⌈Del⌉ or ⌈←⌉ are useful functions. ⌈Clear⌉ is an all or nothing function which forces the user to re-start an entry from the beginning. This is of course frustrating if only the last keypress was incorrect. ⌈Del⌉ is more convenient, as just it deletes individual keystrokes and is easier to control. Unfortunately, As we saw above, ⌈Del⌉ is where a lot of problems of premature semantics come in and creates strange special effects a user has to be aware of — and of course users probably are not aware of these special cases, especially when they are trying to correct their own errors.

The key ⌈Del⌉ can have one of two meanings: delete the last (rightmost in Roman based systems) character in the display or delete the last keypress pressed on the keyboard. In text entry, these two meanings almost always coincide. If we type ⌈A⌉ ⌈B⌉ ⌈C⌉ the display will show ⌈ABC⌉. Clearly, here

16

the last key pressed was `C` and the rightmost key displayed is the same, namely `C`. Pressing `Del` will delete it, leaving `AB` in the display, which seems perfectly obvious.

Yet because of the premature semantics, this is not how things always happen.

When a calculator blocks a second decimal point, pressing `Del` cannot delete the last keypress because the last keypress had no action visible in the display. Depending on how the calculator is implemented, it may delete the rightmost character or do nothing or delete a digit, or even something else. Premature semantics is the source of problems described in section 3.1 of this chapter.

Or consider when a user presses the negation `±` key on a calculator. When a number changes sign, the negative sign appears (or disappears) as a prefix in the display and so deleting the last keypress should remove the negative sign but deleting the last character shown in the display should remove the rightmost digit. You can't have it both ways. Also, if there is only one digit (possibly a zero), how can it be deleted when the calculator always displays a number? These choices have very different effects, as the following table makes the clear:

Time	Key	Display after each key press		
	`C`	`0`		
	`3`	`3`		
	`±`	`−3`		
	`Del`	which one?	`3`	*delete the most recent key press*
			`−0`	*delete rightmost character, and turn 'nothing' into zero*
			`−`	*literally delete rightmost character, but this isn't a number*

And indeed, from the point of view of the calculator, it is not possible to know which is actually required.

A current example is the Apple iPhone calculator (early 2017): the following sequence of user keystrokes will make it crash `AC` `±` `Del` `±` `Del`. The sequence of keystrokes has a simple story: the user keyed `±` by mistake and tries to correct it, but that doesn't work, so they try `±` again (changing sign twice should cancel it out); that doesn't work either, so they hit `Del`. Whoops, the calculator crashes. This is a bug — actually, it's the third bug we've seen in three keystrokes. Apple have some of the world's best programmers, but evidently they do not always make reliable interactive programs, even when they are this simple. We should not be blaming Apple; we should be wondering why it has taken us so long to notice that programming interactive systems is very hard. And the consequence of it being so hard is that it is rarely done well.

Now, summarising this to pull out the morals:

- Programmers add simple features, like entering numbers.

- Programmers add more features. Decimal points and change sign are very simple examples. The `Del` key is another example.

- Some features will be added implicitly. Most number entry systems impose a length limit on how long a number can be.

- Individually and separately, each of these features makes a lot of sense: indeed, they have sensible semantics. Unfortunately, *collectively* the individual semantics don't work together, and we now see that giving each feature, such as the decimal point, a semantics earlier was premature. Most design choices turn out to be inconsistent with later ideas — though the inconsistencies often lie unnoticed.

- The accepted wisdom of iterative design exacerbates these problems. Building a system and then evaluating it with users to see if it needs improving, and if so how, is conventional wisdom. But doing this creates many opportunities for premature semantics to cause problems. New features will be added (or earlier features modified), and the deep implementation, the premature assumptions, of the original features will come back and haunt the later design iterations.

5.2   Construction in forms

These problems of incremental construction are not only seen in calculators of course. Online forms are a common and very useful means of gathering a rich set of data, for instance, name, payment and delivery details in online shopping. The data needed is structured but has several distinct components and online forms, by analogy with paper forms, indicate what is needed and essentially marks up what each part of the entered data should be (for instance, keeping your age and the number of tickets you want to buy date separate data items). Online forms increasingly facilitate correct entry by enforcing the correct semantics for each separate part of the data. For example, rather than entering a date which has many different and inconsistent formats, the user is asked to select it from a calendar. Or the user will at least be cued to enter the correct format by prompts such as "DD/MM/YY." And rather than entering an address which can have even more inconsistencies and ambiguities, people filling out the form can be asked to enter a postal code and then select their correct address from within that list.

However, large forms, such as tax returns, passport applications and so on, that go over several pages often require that each page is correct before allowing the user to proceed to the next page. Yet users may have perfectly valid reasons to move on before they have finished each page.

It is now possible in the UK to order a passport online. The form asks for several separate items of information before asking for a digital photograph. This must, understandably, have very specific formatting and styles which you are not informed of until you reach that page and have already entered half a dozen other items of information. But you cannot go past this page or save your form — because it is not yet correct — so if you do not have the photograph ready, you have to go away and start again when you come back (the form will have timed-out for security reasons).

This is premature semantics during construction. The form is requiring that it is always correctly filled out (and certainly must be if it is to be saved), including requiring that all fields that need completing have indeed been completed even though later pages still have completely unfilled sections. Paper forms of course cannot enforce such premature correctness, allowing users a much more flexible workflow.

It is worth noting that the UK Government has taken usability of its systems very seriously and has made a lot of effort to make all its online resources as accessible as possible to as wide a portion of the population. Thus, even when usability is taken seriously, it is still possible to end up with problematic interactions because of premature semantics. As we've said, we think it is a shame that HCI (and UX in particular) has become quite separated from computer science. You cannot have good UX (or good HCI) without a foundation of good CS.

5.3   Problems of representation

If an item of data must always be valid, there is also the problem of what to represent in the display. Early calculators solved this by having physical displays that necessarily had inflexible constraints. They could not display more than a fixed number of digits and often not even digits themselves very well [22]. Software calculators have no such obvious constraints. Once again, consideration of the semantics through FSMs helps to clarify the problems that arise.

Mathematically, there is no constraint on the size of a number so the representation can be literally any finite length. The FSM would therefore permit digits to be appended to the end of a number so long as users keep pressing them. This is one behaviour seen in section 3.1 but of course

it can result in an unreadable display — this is a logical necessity due to the finite limitations of any device together with the unbounded possibilities of what a user might enter.

The alternative is that the FSM prevents numerals from overflowing the display. That is, when a user reaches a certain length of number, no further digits are added at the end. Now however, the last digit displayed is now not the same as the the last key pressed, and so ⌐Delete⌐ becomes ambiguous causing problems in construction.

There is also the boundary case seen where the last key pressed in a long number does not result in the correct digit being displayed; this happens when the number displayed is almost at the maximum value the calculator's internalarithmetic can handle. What this makes clear is that there is no obvious FSM or regular expression underlying number entry in such a calculator. Instead, the number entered has been re-represented internally in some other format, probably a binary representation of the numerical value, and so results in rounding errors when displayed back to the user. This is premature semantics at its worst: the meaning of the number is altered even before the user has finished entering it!

In some cases, problems of representation can also become problems of construction. Modern calculators now have the capabilities to represent numbers correctly in more sophisticated forms, such as standard exponent form, such as 6.022×10^{23}. Early calculators that used seven-segment displays could not represent this format correctly so instead they compromised with a notation like `6.022E23`, with E standing for Exponent (i.e., the exponent of a power of ten, so E23 means what we normally write as 10^{23}). In some early scientific calculators, a special key allowed users to enter this notation but not directly. Oddly, modern calculators, such as iOS calculators, display numbers in this historical and outmoded format: thus the iPhone (iOS v7.1.2) calculator displays `6.022e+23`. Not only is the representation a historical peculiarity, it interferes with construction. The ⌐EE⌐ key (not ⌐e⌐!) in the same calculator allows users to enter numbers in this exponential form but acts like an operator (like ⌐+⌐) and ⌐=⌐ has to be pressed to find out what it has done. This means ⌐Del⌐ cannot be used conventionally: correction is not possible without a complete clear and re-entry! It looks like the programmers have not thought this through, yet they think "it works" (and probably passes all their tests) — that is, it's premature semantics at work.

5.4   Premature semantics more generally

We want to emphasise that our discussion of premature semantics is very general. Nevertheless, it is easy to continue illustrating it as it applies to numbers — we are familiar with numbers, they have a simple semantics, and we know what they should do without further explanation here. A simple version, then, of premature semantics is

1. The user's task requires numbers;

2. The programmer implements a program using numbers;

3. These numbers will typically be declared in the program as integers or floating point types, which is how programs implement numbers.

4. This commitment to integers or floating point numbers, as may be, is often premature, since it limits the language the user interface can implement. It does not cater for the user entering things that cannot be parsed as integers or floats. It therefore creates problems, possibly very complicated problems. We have shown that the problems are hard to spot, hard to describe, hard to for the user to correct — and have persisted for years.

Although we focused on numbers because we all understand them very well, more generally, of course, user interfaces generally support broader tasks than can be described as ending up with numbers alone. Here, object oriented programming (as one popular approach) provides a far wider range of types than integers and floats: indeed the programmer can create new types. So if the

user interface needs drawings, an objected oriented program will have a type `drawing` (say) that implements what the programmer thinks drawings are.

The premature semantics issues still arise. The user wants drawings, and the program implements drawings. Job done! The user wants x, and the program implements x. Job done! But we saw, at length and exhaustively above, even if x is something as very well-understood as numbers, it still goes horribly wrong.

Acknowledging types can help implement anything, let's look at integers for concreteness; integers are a type of number. The user wants to end up with an integer to do whatever they want to do, for instance to enter their credit card security number. The program wants to implement this number so that the program can work; typically the programmer will declare a variable to hold this number, probably declared as an `int`, which is the common abbreviation for integer type in programming languages. Here, then, are some complications:

- Security numbers typically have three decimal digits, such as 479.

- The programmer will declare a variable, probably saying `int code` or similar to store the code. However the `int` type typically covers whole numbers from minus 2,147,483,647 to plus 2,147,483,648.

- Therefore the programmer *may* provide some sort of programming, formally called subtyping, to make their program check the user's number is exactly three digits. They probably won't do this, because 000 to 999 *obviously* fits within an `int` without special programming.

- The programmer will make some decisions over leading zeros. How will they handle 057, for instance? Is this a two digit number (equal to 57) or a three digit number with a leading zero? Unfortunately, in the program 057 and 57 are the same.

- The programmer will optimise their program, doing bit operations, writing in assembler, and more. Optimisations make programs more efficient, but they do so by relying on *assumptions*; and the more assumptions a programmer makes the more likely they are going to make premature assumptions further embedding in the premature semantics of those assumptions.

These sorts of issues are familiar to programmers, but if you agree with them you have already fallen into the trap of premature semantics!

- What do we do if the user does not enter a security code? This is **undefined**, and undefined is not a value an `int` can represent. Therefore the programmer must do a lot of work to handle this case, as no programming language helps directly. Many quality programming languages, Java being a good example, *force* the programmer to ensure that their variable `code` is defined — but it is possible that the user does not define it. If it is defined, it will have to be a number like zero (or 000) but that is not what the user did if the user has not entered it. This is a conflict.[5]

- What do we do it the user enters an invalid code, perhaps their credit card's expiry date 11/19 by mistake? This is an **error**, and error is not a value for an `int` either.[6]

These are both fundamental problems created by premature semantics. If either undefined or error occurs, what should the program do? Can the user still save their form they are entering the

---

[5]Forcing variables to be defined reduces the risk that programs will crash, but it does not increase the chances the programs do what they should. Here, programming language designers (and their compilers) force premature semantics on users of their languages. Programming languages like Swift allow variables to be undefined, and they rigorously check how programmers managed undefined values; this helps.

[6]No programming language handles errors well — it is still a research problem. This is why many user interfaces say "Unexpected error" — if they had correctly expected the error they would have fixed it!

credit card details in? Even though saving a form makes perfect sense — especially when the user has made a mistake they want to think about — the program (as usually written) cannot offer the service, because undefined and error themselves cannot be saved as integers.

The programmer thought three digit security codes could obviously be implemented as an `int`, but — typically later, sometimes even after they have left for another job — somebody has to start hacking their program to handle "special cases" to keep the usability professionals happy. These well-intentioned iterative "improvements" are then likely to lead to further bugs.

> *Note.* When we explain a problem as we did above, not only did we choose a problem that was simple enough to explain but you have the privilege of hindsight: those errors we described seem very obvious and easy to avoid, don't they? But if you think you could correctly implement a three digit security code (thanks to our lucid explanation), pause to think how many people in the world thought they could implement years — how hard could a four digit number be? Yet we ended up with one of the world's largest computer fiascos, the Y2K problem.

Figure 4 shows what still happens in 2017 — almost two decades after Y2K. Here a programmer thought that a to do list was a simple database, then they thought they would allow the user to run their to do list from several places (their desktop, their mobile phone, their tablet). And then they discovered that they had not implemented distributed error correctly. Their program just drops out in a panic, leaving the user an unanswerable question. Note the premature semantics: Omnifocus prematurely decided to program *perfect* databases (analogously to calculators implementing "perfect" numbers but not supporting what users do). For whatever reasons, the programmer's utopian (premature semantics) databases do not exist in the real world of the user's actual tasks and activities.

Unlike a calculator, Omnifocus at least notices when its premature semantics fails, though it does not do this very gracefully.[7] A user knowing they have a problem is a step better than a nurse using a calculator for a drug dose and *not knowing* there is a problem.

5.5   Non-numeric premature semantics

As discussed earlier, any user interface constrains the possibilities of interactions and so defines a language for interaction. Some of these constraints have become so familiar that we do not necessarily see them any more — earlier (section 3.1), we speculated whether this was due or at least partly due to hazing.

For example, when you create a new blank document in a word processor, such as Microsoft Word, you are presented with a blank sheet of paper (well, a virtual one). However, this sheet of paper does not work like real paper in that you cannot put your cursor wherever you like and start typing. In fact, there is only initially one place the cursor can be placed. It is only when text (including spaces) has been entered can the cursor be repositioned, but even then it has to be between characters and not in blank spaces where there are no characters. This is not necessarily bad premature semantics: we do not think many users would these days expect any different interaction. But at the same time, it is an unnecessary constraint where cursor position is inherently tied to the textual content of the document. This contrasts with new opportunities for interaction where with that very same document on a Microsoft Surface, it is possible to use the pen to make "ink" marks anywhere on the page. So even if the semantics of the cursor language for the original Word are appropriate, they are not consistent with the pen language. Interestingly, this problem was solved in the 1980s [23], but hazing or other reasons meant the solution was never adopted.

---

[7]We are left wondering why it does not say: here is a list of incompatibilities, which do you want to keep? Sorting out to do items, which is what this question amounts to, is what the app is all about! So what is the problem? Why not a "to do" project which is pre-populated with conflicts the app can't be bothered to sort out itself? It already has the user interface to do this, so in principle it would be trivial to rescue the user. Unfortunately, premature semantics is too often invisible to programmers, even when they have the generosity to point out (here, in dialog boxes) when it fail the users. Perhaps Omnifocus's UX people believe the programmers when the programmers say they cannot do any better?
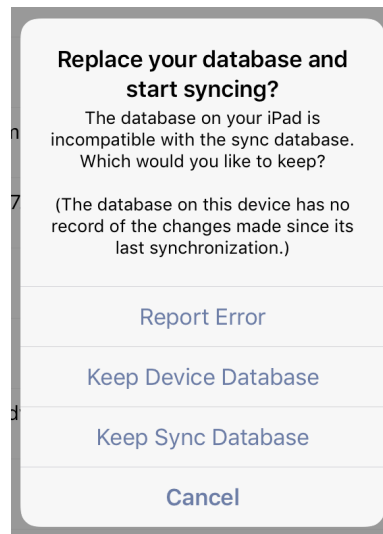
Figure 4: What the "to do list" app Omnifocus does when it encounters an error it had not anticipated in its premature implementation. It successfully detects an "error" but has no idea what to do. Note that the dialog box does not give the user sufficient information to make a wise decision either; whichever option they choose, they risk losing information. Ironically a "to do list" is supposed to help users remember what to do, not help them forget things!

Anyone who has used a self-checkout at a supermarket will be aware of the frustrations they can present. One particularly interesting example is the premature semantics in the process of bagging. The assumption on the part of programmers is that people in the UK use the shop's lightweight plastic bags and so it is a simple matter to check the weight of items as they scanned and bagged to ensure that customers are honest when bagging up their goods. However, with new laws in the UK that charge for shop bags and so encourage customers to use their own bags, many checkouts are not setup to deal with bulky shopping bags or rucksacks that first have non-negligible weight and secondly do not fit nicely into the bagging area and so onto the built-in scales. The result is almost always an error ("Unidentifed item in the bagging area") when you try to be a good citizen and user your own bags.

This is not so much a case of premature semantics but more like evolving semantics where, over time, the expectations and assumptions of the programmer have been superseded by social changes that could not reasonably have been anticipated. And while people have changed, the language of the interaction has not changed with it and this thus leads to problems. In our 20/20 hindsight the semantics was premature.

5.6   Teaching HCI

We both teach HCI, and one of us does it by giving an overview of the topics and concerns in HCI, then allocating students to give brief lectures on each topic. This creates two problems that computers are good at solving. First is matching students to topics as they change their minds and negotiate — this is part of the course teaching, to understand the variety of topics and their relationships to each other. Secondly, we have to order the lectures sensibly and allocate each student a slot in the timetable to present their lecture to the class. These are both standard computer science problems: a simple case of stable marriage and topological sorting.

This year, one student changed their mind after the timetable had been fixed and the student lectures had already started. The student had a better idea about the topic for their lecture. No problem! Update their preferences, rerun the program . . .

Unfortunately, almost the entire timetable changed when the topological sort was re-run. This upset a few students who had started preparing their lectures (it mostly upset the students who were postponing preparing their lectures to the last possible moment that the original timetable allowed).

The problem was accidental premature semantics — the imposition of the first ordering before every student had settled their topic choices.

A priority in our semantics was that the order of the lectures must make pedagogical sense, for example understanding why user centred design is important will precede a lecture on iterative design (you don't need iterative design if you don't believe in UCD).[8] But when one student changed their preferences, almost all timings needed to be changed.

So next year, the program will allow students to change their preferences, but once a timetable is published, students will be allowed to change their preferences (when they have good reasons, exploring which is part of their learning) *provided that* the rest of the timetable does not change. Currently, the program has no persistent idea what the timetable is — it is recomputed every time the program is run — so there is no semantic concept of "not changing the timetable" since it does not even remember the last one. This premature semantics clearly needs fixing.

One advantage of teaching HCI is that any problems lecturers and students encounter readily translate into HCI learning opportunities. Here, we had a nice case study in iterative design and how designers (in this case, the lecturers) need to see how their users (here, the students) experience the consequences of their program designs.

From the programmer's original perspective, the program did what it was supposed to do. It took good account of student preferences. It generated a nice timetable. To get it to work, it had to have some semantics, which seemed pretty obvious to the designers. Moreover, the original program worked very nicely. Only later did it become apparent that there was a use case which raised serious problems. The original semantics of the program were wrong: they had been premature.

5.7 Drug dosing

Another example is that a form may want a number. If the user does not provide a number, premature semantics "creates" a number. Hence the application gets zero (probably) but the user entered nothing. Zero and nothing are not the same things — for example, zero is a number, but nothing is the absence of a number, so can't be a number.[9]

This matters — for example, a drug dosing application cannot tell whether a nurse has checked that no drugs should be given to a patient, or that the nurse has not checked but the absence of data has been turned into a zero. In both cases the application thinks zero, but only in one case does it really mean zero — and the patient may get the wrong treatment.

5.8 The tip of the iceberg

It is possible to expand this list of premature semantics examples innumerably as problems of interaction are mapped to the meaning people are trying to convey through there actions. However as our examples show, sometimes the problems arise because of premature oversight (the main cause, we think, of premature semantics), or the development of technology, sometimes because of contextual or cultural developments, and sometimes because the accumulation of otherwise sensible features that interact with each other incoherently. It is currently hard to imagine how to define a language involving multiple physical objects like a supermarket self-checkout. But if there

---

[8]If topic a comes before c, and topic a must follow b, and topic d is independent, then a topological sort finds possuble orders dbac, bdac, badc, bacd.

[9]Putting this another way may help: if we write numbers between brackets, then [235] represents the number 235, [0] represents the number 0, but [] — that is, nothing between the brackets, is *not* a number at all. Premature semantics "solves" this by simply saying anything between brackets must be a number even if it isn't.

were such a language, then it may be possible to formally analyse such systems to see how the changing world interacts with the changing language.

6 Solving the problem of premature semantics?

Addressing premature semantics needs to detach the user interface, particularly user input, from the semantics that implements it. In many ways, this is the approach of many of the old-fashioned command style interfaces of early computers. A user could freelyenter any command and its parameters. It is only when they pressed ⟨←⟩ (i.e., enter, return, line feed, etc) was the command interpreted; if it was invalid it was rejected, sometimes even with a helpful error message. This style of interaction has steadily been left behind because of the burden it puts on users to get things right all in one go [16], something we know can be challenging even for the most diligent users [10]. Instead, what is needed is a form of data entry that can be done alongside semantics but without constraining what users do — but also guiding them into correct semantics. Forms are an example of a user interface paradigm that does do this (to varying extent, depending on the quality of the implementation) but they are not necessarily free from premature semantics (see section 5.2).

In order to balance semantics and free-entry, Thimbleby [27] implemented a system for number entry (and other forms of data entry) using a traffic light coding. A user is free to enter data however they wish but as each key is pressed the system also processes the input using an underlying FSM for the correct semantics. A user can press any keys but if it leads to a semantically invalid value, say if a key press adds a second decimal point, then the display turns red to indicate this has no meaning. Conversely, if the user has entered a sequence of keypresses leading to a semantically correct item then the display is green to indicate that what is currently displayed is semantically correct. Finally, the display will be orange to display that it could be the start of a semantically correct entry but needs further correct keypresses to make it so; for instance, if the display is `1.` it needs some decimal digits before becoming a valid number (at least if we want to avoid "naked decimals" [7]). Finally, the user interface can make sounds (or it could provide tactile feedback) when it transitions between traffic light colours: the user does not need to look at the display.

With this approach, the user is constantly aware of the potential or contingent semantics of their entry but without being constrained at all in their entry or correction. The interesting thing about this traffic light approach (and a range of related variations to better suit different sorts of user interface) is that the light colours can be automatically determined from the underlying FSM.

Such a traffic light system uses FSMs but does not work only with an FSM as the display itself acts as a buffer to store the user's actions, so ⟨Delete⟩ etc correctly work on the buffer regardless of its semantics. The system does not accept the user input, as represented on the display, until some form of commit key is pressed and that can only be pressed when the display is green.

6.1 Reengineering

If programming user interfaces causes so many problems for programmers, another approach is to consider if the designs themselves are faulty in some way, rather than the programs that implement them. Certainly, computers cannot correctly implement something that is inconsistent.

We should consider stepping back entirely from traditional user interfaces dictated by historic and outmoded technology. Why reproduce mechanical calculators, which inevitably were compromises when now computers can do anything? We should start by asking questions like "what is the purpose of a calculator?" — or "what is the purpose of the system that is intended" — we should obviously worry about all UIs, not just calculators!

A calculator is used to perform artithmetic calculations that arise in real situation with numbers or calculations that are difficult to do in the head or on paper. Some calculations fit well with entry into traditional calculators: how many oranges have I got if I have 23 crates with 35 oranges
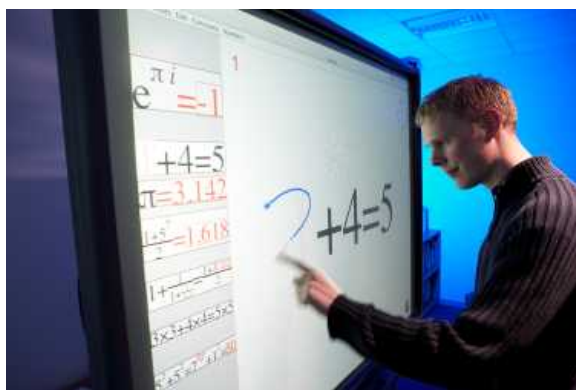
Figure 5: The classroom version of an intentionally *not* premature semantics calculator. The calculator also runs well on handheld tablets.A photograph of a user interface cannot do the interactive experience justice; the person shown using this calculator is Will Thimbleby, the implementer.

per crate? This problem can be translated into a sequence of key presses that look roughly like 23×35. However, many calculations we want to do are far less direct: if a worker works 50 minutes on and 10 minutes off every hour for 7 hours, and each of their data entry tasks takes on average 35 seconds, how many data entry tasks will they get through in a typical working day? Normally, you'd solve this with a calculator by either using pencil and paper to write down the problem and then converting to a calculation or chancing your luck by a good guess as you work on the calculator. A better solution would be to represent the problem as you understand it in the calculator and then work with the calculator to get the answer. This is precisely the philosophy behind a novel gesture-based calculator [28].

This novel calculator avoids the problem of premature semantics by allowing the user to write whatever they want on the calculating surface. The calculator was designed to allow the user to enter anything, even nonsense, and therefore does not have premature semantics — it does not require or force the user's input to be correct in any way, but uses another colour to correct the user's input. It turns out that correcting the user's input *also* provides the answer to the user's calculation. The relation between correction, error and premature semantics is profound and worth explaining in more detail.

You may want to know what to multiply 2 by to get 6 (you may well want to do something harder, but we are interested here in the principles not the actual problem). On a normal calculator, you might key `AC` `6` `/` `2` `=` to get the answer you want, which here will be 3. However, note that you have had to convert the original question using multiplication into a question the calculator can handle, using division.

With the new calculator, typing "$2\times\ = 6$" is not correct, but would be corrected to show the answer `3` in the right place: $2 \times$ `3` $= 6$. In other words, the flexibility of correcting anything significantly increases the power and flexibility of the calculator.

Or if the new calculator was used like an old one, typing "$6/2$" is also not correct as it stands, so it would be immediately be corrected to "$6/2$ `=3` ." If the user continued typing "$6/2 =$" that is fine too, as it will be corrected to "$6/2 =$ `3` ". Indeed, if the user typed "$= 6/2$" (i.e., the other way around) that would be corrected to " `3` $= 6/2$".

See figure 5. The calculator interprets whatever the user tries to enter so if the user cannot represent it, the calculator doesn't make it worse. And the result of the calculation and the calculation itself are co-present and persistent, which means that the user representation can be re-constructed as necessary and even the calculator representation can be re-constructed.

This is a specific solution to the problems of premature semantics that works with arithmetic

calculations (which is really useful if that was all it did). However, it does show that really understanding what the user needs can lead to much better systems than the blind adherence to existing ways of doing things that computers, or at least programmers, may struggle with.

It is interesting how many applications — from mobile phones to word processors, spreadsheets to drawing programs — need numbers but cannot cope with arithmetic! How wide should your margins be is a simple question for a word processor byt often needs a calculator to solve it: the calculation is something like the width of A4 paper is 21 cm, a good line length is 6.5 in, so the margin is $(21 - 6.5 \times 2.54)/2$ — but very few apps allow such calculations. Apps prematurely want numbers, which is how they prematurely implement the user's problem, here, $(21 - 6.5 \times 2.54)/2 = 2.245$, so the programmer thinks 2.245 is good enough (why do something when the user can?) and all the user is allowed to do is enter the number, not an expression. Such "programmer thinking" — premature semantics requiring numbers rather than calculations — makes the user's work harder.

We don't actually believe programmers *really* think of ways of making the user's life harder (except in some games). What they do is unconsciously make their own lives easier — premature semantics — and that has consequences they are often unaware of. It is far easier to read a number than read an expression, and since a number *is* needed the programmer assumes that is all the user interface needs. This is premature semantics of course.

Clearly, the user interface ideas of this calculator are much more general than calculators. Usability studies of this calculator were very encouraging [28, 29], so it is not just a theoretical discussion about premature semantics.

6.2   The semantics of interaction

It is well-known in mainstream Computer Science that compiler correctness is a critical problem; many compilers have subtle bugs, and finding and proving they are free of bugs is a hard research problem. However, once formal methods are used, one can create improved compiler construction processes that avoid entire classes of bugs, as well as construction processes that better match the capabilities of formal methods tools. The goal of such research is to make compilers more dependable, and hence ensure that hardware reliably does what programmers want it to do. In short, compilers are designed to be semantics preserving.

Similarly, a user interface makes hardware perform as the user intends, as expressed through a sequence of interactions with the computer. This is analogous to a compiler that makes hardware perform as the programmer intends — except a programmer inputs a static string (the program source code) whereas the user inputs a sequence of actions, including keystrokes and often two-dimensional gestures, such as taps and strokes, typically using their finger or a mouse (or even 3D in VR, speech and so on). A critical difference is therefore that a program is textual, and a programmer can think about it in various ways, for instance writing it down on paper and reasoning mathematically about it. In contrast, a user interface input almost always has no simple concrete representation, and certainly nothing that could be readily written down on paper and reasoned through.

When a bug is found in a compiler, it is usually very easy to reproduce the bug. A specific fragment of program source code does not do what it should, and it is generally possible to write a program where an assertion fails when the compiler bug causes an incorrect result, and so on. In contrast, when a bug is found in a user interface, reproducing it requires considerable attention to detail because it is not easy to represent and reproduce the interactions that revealed the bug. Indeed, the user who found the bug is unlikely to remember exactly what they did, since the effect of the bug cognitively interferes with their memory of what they are doing.

Seen from this perspective, any interaction with a device requires the preservation of semantics: the meaning that the user intends by their actions is correctly represented in and acted on by the device. At the same time, the fallibility of users needs to be recognised so that not everything a user does should be acted on immediately. In some contexts, such as text entry, this may not be a big concern but in other contexts, particularly ones that are mission critical or safety critical such as

aircraft cockpits, it is.

However, historically, there has been little concern for examining the semantics of interaction. To really address semantics and reason about semantic correctness requires the use of what are known as formal methods, of which tools like FSMs and regular expressions form a central part. However, there is virtually no research in formal methods and user interfaces; it is very much a niche concern. In fact, formal methods are often considered counter-productive since even a formally correct user interface may not lead to one that is acceptable to users. Thus, much emphasis in evaluating interfaces is in the experience of users not the correctness of their interactions.

Another cause for the neglect of formal methods comes from a pragmatic view of what it takes to produce user interfaces. User interfaces are "easy to program" and interaction is an emergent property of executing a program. The formal properties we might want to think about even for apparently simple devices like a calculator, such as display overflow and the regular expression for valid numbers, seems like a long way from the simple tasks that most users would encounter. Moreover, the list of formal properties have not ever been specified, even for calculators which in principle do have formal semantic foundations. Diligent user interface programmers would struggle to formally evaluate a user interface even they wanted to.

And at the end of the day, much user interface design is about the acceptability of interfaces to users. Most users are tempted to say "the user interface works" — why worry about a quirky case, which anyway can be avoided in a work-around if the bug ever affects a user (assuming the user notices the bug)? In addition it is well known that conventional informal user interface development (such as focus groups, scenarios, personas, etc) is a critical tool for user interface acceptance. Thus, even what we would identify as bugs in the semantic interpretation of an interaction, users frequently dismiss as not relevant.

6.3 Mature semantics

Moving on from the current loose thinking about user interaction requires several changes that encompass both research and culture. First must come the recognition amongst programmers, but also more importantly among users, that the process of interacting with a system has meaning. And that meaning is perverted or lost in many existing systems. A person who engaged in such wilful neglect of meaning would be considered a sociopath but we accept such computer systems most often with rolling of eyes. There are many comedic sketches that capture the problem, like "Computer says no!" [32].

Users need to recognise the importance of preserving meaning in interactions but also that, as with people, this may come with an overhead of interaction that is not quite so quick or occasionally requires a bit more effort on their part. To support this, programmers need to be able to develop interfaces, like the traffic light number entry system, that first are able to support semantics and secondly that can be reasoned about through the use of formal methods.

This last point leads to the third change that is needed, which is more research on what it means for interaction to be semantically correct. We know compilers have bugs; do user interfaces? Do user interface bugs matter? In other words, is research in user interface bugs useful? To address these questions we have the additional problem that user interfaces are not or very rarely formally specified, so the concept of preserving semantics is an awkward concept even before we start! In some cases, like number entry, we are beginning to understand some of the formal properties an interface needs [24, 2] and to develop the tools to address them. With other areas like setting up an infusion pump or controlling a semi-autonomous vehicle, we are a long way from even specifying what the semantics should be but such devices already proliferate and are causing problems due to the breakdown of semantic correctness.
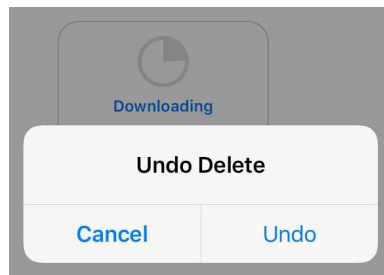
Figure 6: A familiar dialog box for users of iPhones, but what does it mean? Does pressing "Cancel" mean cancel undo, which means delete, or does pressing "Undo" mean undo undo delete, which probably means delete? So pressing either button means delete? We imagine — we hope — the programmer and other privileged people know what the choice means, and (presumably!) what these buttons means was obvious to them and (presumably!) they mean different things (otherwise why give a vacuous choice to users?). In any case, what exactly are we accused of deleting or undoing deleting that we might want to cancel or undo? Presumably (!) the programmer knows, but they haven't bothered to tell us to help us out of our confusion. Did the programmer ever employ an HCI expert to run a usability test to uncover and negotiate the curse of knowledge that has undermined the clarity — premature semantics — of their programming? (Note that the probability of finding bugs like this during user testing is low, another argument for using formal methods.)

6.4 The curse of knowledge

A profound problem with premature semantics, the reason why it persists and has not previously been recognised, is that *it makes so much sense*. The programmer prematurely committing to a floating point number (calculators) or to an integer (credit card numbers) or to a perfect database (to do lists, web forms) all seem like very fine decisions — much like all the other decisions programmers have to make. If programmers did not make such decisions they would be paralysed and nothing would happen.

Steven Pinker's terrific book *The Sense of Style* [13] on writing and how to understand writing well discusses the **curse of knowledge**. When we write, we know what we mean, so we tend to write complex sentences and complex paragraphs — what they mean is quite obvious to us, because we know what we are thinking. Our readers on the other hand, do not know what we mean and they have to decode (parse and interpret) what we write to find out. As writers we think this "decoding" is easy — in fact, we already know what we mean so we don't need to do any decoding to find out! Why are our readers having trouble?

This is the same problem designers and programmers have; instead of writing English, they are programming interactive systems. They, too, have a curse of knowledge — they know what their systems are supposed to do, so their systems seem much easier to them than they really are. (HCI has developed techniques that programmers resist to fix this problem.) Figure 6 gives a simple example.

Premature semantics makes sense to programmers and, like the curse of knowledge, it is obvious that it makes sense. Note that "making sense" is another way of saying "it has the right semantics." Hence because of the curse of knowledge programmers do not notice they are creating problems for users. So, Steven Pinker will help us write better English for readers; he, properly understood, will also help us design (i.e., write programs for) better interactive systems for users.

7 Conclusions — moving towards mature semantics

In this chapter, we have considered in some depth the apparently simple task of entering numbers into an interactive system and shown that it is still a surprisingly poorly understand task in interaction design. Premature semantics brings into relief the complexities of such seemingly simple tasks. This motivated our more general discussion of premature semantics, for it applies to all forms of user interface.

With modern user interfaces, data entry is no longer like writing with a pen on paper but is a dynamic, fluid, erroneous and correctable process. Until such processes are deemed complete by the user, it is very risky to assign semantics to what is being entered. Yet without guidance on the semantics, users may end up in a mess. We think our examples of numbers are so basic and widespread — even "trivial" — that they convince us that they can't possibly be the only problem with user interfaces. They are typical of a whole range of poor user interface design — and why it happens. If something so trivial and apparently well-understood is broken, what else is?

Programmers do not have or prefer not to use tools to systematically identify and address these problems. User Experience practitioners did not (until this chapter) have the vocabulary to describe them, and virtually no vocabulary at all to talk in the same languages as programmers need to understand them.

We hold that what is needed is systematic research on formal methods for interaction that allow all concerned to specify and reason about the semantic properties of interaction.

User iterations will change in the future in a way that will affect users (e.g., with implants) profoundly but we would hope that users and HCI workers will see the benefits of being able to avoid premature semantics that limits the user and induces complex errors they cannot fix. There is a better future.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques*. Addison wesley Boston, 1986.

[2] P. Cairns and H. Thimbleby. Interactive numerals. *Royal Society Open Science*, 4(160903), 2017.

[3] Stuart K. Card, Thomas P. Moran, and Allen Newell. The keystroke-level model for user performance time with interactive systems. *Communications of the ACM*, 23(7):396–410, 1980.

[4] Asaf Degani. *Taming HAL: Designing interfaces beyond 2001*. Springer, 2004.

[5] J. B. Gasen. Support for HCI educators: A view from the trenches. In *Proceedings BCS HCI Conference*, volume X, pages 15–20. Cambridge University Press, 1995.

[6] Jeremy Gow and Harold Thimbleby. Maui: An interface design tool based on matrix algebra. In *Computer-Aided Design of User Interfaces IV*, pages 81–94. Springer, 2005.

[7] Institute for Safe Medication Practices. *ISMP List of Error-Prone Abbreviations, Symbols, and Dose Designations*. 2015.

[8] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2nd edition, 2011.

[9] Thomas K. Landauer. *The trouble with computers: Usefulness, usability, and productivity*, volume 21. Taylor & Francis, 1995.

[10] Simon Y. W. Li, Anna L. Cox, Ann Blandford, Paul Cairns, Richard M. Young, Aliza Abeles, et al. Further investigations into post-completion error: the effects of interruption position and duration. In *Proceedings of the 28th Annual Meeting of the Cognitive Science Conference*, pages 471–476, 2006.

[11] Robert M. Mulligan, Mark W. Altom, and David K. Simkin. User interface design in the trenches: Some tips on shooting from the hip. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI'91, pages 232–236, New York, NY, USA, 1991. ACM.

[12] Kai A. Olsen. The $100,000 keying error. *IEEE Computer*, 41(108):106–107, April 2008.

[13] Steven Pinker. *The Sense of Style: The Thinking Person's Guide to Writing in the 21st Century!* Penguin Books, 2015.

[14] Yvonne Rogers. HCI theory: classical, modern, and contemporary. *Synthesis Lectures on Human-Centered Informatics*, 5(2):1–129, 2012.

[15] Dina Salah, Richard F. Paige, and Paul Cairns. A systematic literature review for agile development processes and user centred design integration. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering*, page 5. ACM, 2014.

[16] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction design: beyond human-computer interaction*. John Wiley, 2007.

[17] I. Sommerville. *Software Engineering*. Pearson, 10 edition edition, 2015.

[18] H. Thimbleby. Calculators are needlessly bad. *International Journal of Human-Computer Studies*, 52(6):1031–1069, 2000.

[19] H. Thimbleby. Safer user interfaces: A case study in improving number entry. *IEEE Transactions on Software Engineering*, 41(7):711–729, 2015.

[20] Harold Thimbleby. User interface design with matrix algebra. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 11(2):181–236, 2004.

[21] Harold Thimbleby. *Press On: Principles of interaction programming*. The MIT Press, 2010.

[22] Harold Thimbleby. Reasons to question seven segment displays. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1431–1440. ACM, 2013.

[23] Harold Thimbleby and Richard Bornat. The life and times of Ded, display editor. In J. B. Long and A. Whitefield, editors, *Cognitive Ergonomics and Human Computer Interaction*, pages 225–255. Cambridge University Press, 1989.

[24] Harold Thimbleby and Paul Cairns. Reducing number entry errors: Solving a widespread, serious problem. *Journal of the Royal Society Interface*, page rsif20100112, 2010.

[25] Harold Thimbleby, Paul Cairns, and Matt Jones. Usability analysis with Markov models. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 8(2):99–132, 2001.

[26] Harold Thimbleby and Andy Gimblett. User interface model discovery: Towards a generic approach. In G. Doherty, J. Nichols, and Michael D. Harrison, editors, *Proceedings ACM SIGCHI Symposium on Engineering Interactive Computing Systems — EICS 2010*, pages 145–154. ACM, 2010.

[27] Harold Thimbleby and Andy Gimblett. Dependable keyed data entry for interactive systems. *Electronic Communications of the EASST*, 45:1/16–16/16, 2011.

[28] William Thimbleby. A novel pen-based calculator and its evaluation. In *Proceedings of the third Nordic conference on Human-computer interaction*, pages 445–448. ACM, 2004.

[29] William Thimbleby and Harold Thimbleby. A novel gesture-based calculator and its design principles. In *Proceedings 19th BCS HCI Conference*, volume 2, pages 27–32. Citeseer, 2005.

[30] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[31] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem: A correction. *Proceedings of the London Mathematical Society*, 43:544–546, 1938.

[32] David Walliams and Matt Lucas. Computer says "no", May 29, 2015.

[33] Richard M. Young. The machine inside the machine: Users' models of pocket calculators. *International Journal of Man-Machine Studies*, 15(1):51–85, 1981.