Keyed data entry
for dependable interactive systems

Harold Thimbleby
FIT Lab — Interaction Laboratory, Swansea University
Swansea, Wales
harold@thimbleby.net

ABSTRACT
Keyed data entry is fundamental and ubiquitous, and occurs when filling data fields in web forms, entering burglar alarm pass-codes, using calculators, entering drug delivery rates in infusion pumps, making cash withdrawals from cash machines, setting altitudes for aircraft flight control units . . . to name but a few of its applications. Unfortunately data entry is often implemented ignoring the possibility of user error, and hence data entry is not fault tolerant. User slips can result in unpredictable and potentially adverse outcomes.

A principle-based approach is proposed for keyed data entry and introduce color coding (traffic lights) to give the user feedback, on a user interface that is correct by construction (C x C). The approach contrasts with conventional systems that are inconsistent and unhelpful to users. We introduce divergence, a certain loss of predictability in a user interface, and show that it is in general unavoidable (though handling it rigorously but tolerantly, as here, is better than ignoring it).

More empirical data and evidence-based theory is needed, though this need not be an excuse to delay engineering improvements.

1. INTRODUCTION
User interfaces have inconsistent and sometimes dangerous ways of handling user error, and this paper argues this problem is soluble within the constraints of keyed data entry. We describe a tool and the approach it supports, which permits interfaces to be rigorously constructed and analysed, as well as consistently implemented across and between applications.

In the context of this paper, dependability is the trustworthiness of a computing system that allows a user to justifiably rely on the service it delivers [2]; this breaks down into predictability (for the user), rigor (for the developer), and appropriate integration between the two—software engineering that results in interactive systems consistent with their documentation (requirements, help, training, manuals, etc). Other criteria include maintainability, since a design will be modified (iterative design requires it).

A “dependable user interface,” then, means that developers have indeed implemented the user interface (UI) they intended, and users can interact with the UI to dependably achieve their goals. Both developers and users make errors, and dependable systems should be fault tolerant or resilient in the face of these and other faults. Humans will always eventually make errors, and dependable UIs must therefore be fault tolerant to human error. Contemporary UIs for keyed data entry are not dependable, as this paper explores and attempts to mitigate.

The clear statement of a problem helps solve it, and ideally also helps with a family of related problems. A clear statement helps people recognize there is a problem and appreciate that it has or may have a solution. If, moreover, the approach involves a computer program then it can be reused and reapplied to solve many related problems. A clear approach may identify further problems that had not been anticipated, and hence offer routes to solutions to unspoken issues. George Pólya in his How to Solve It [14] makes similar points, though his classic writing predates the computer as a significant player in problem solving—as a tool that encourages the spread and uptake of good solutions. Sadly, computers sometimes provide an inefficient and low quality solution. The aim of this paper is to identify a problem and help move current low quality solutions to a more rigorous and general foundation.

We introduce the term TUI, a tool-based user interface (by analogy to GUI, a graphical user interface), a user interface that is clearly and consistently designed and implemented as an instance of a larger family of related interfaces. We prefer the term TUI to UIMS (user interface management system), as a UIMS typically prioritises flexibility and generality, whereas a TUI prioritises dependability and rigor. Hence TUIs are better for users; carefully considering the general and abstract properties of user interfaces at the tool-level leads to better UIs. Consistency, easier iterative design, improved dependability and so on are natural consequences of a good tool-based approach.

All generated TUIs share crucial interaction properties, user experience (UX), and hence—if nothing else!—considerably reduce transfer errors and improve user learning times between devices and systems. Additionally, TUIs are easier to modify while maintaining clarity and hence support iterative design well.

For clarity, we will use the following typographic notations:

- abc program code developers write
- [A X | Z C] keys users press to enter data; in this paper, we do not complicate discussion with combinations of keystrokes such as [SHIFT][A]
- [CANCEL] keys represented with a full word represents a single key users press to control interaction
- ABC symbols displayed on a screen or panel intended for the user to read. When a display is shown in this paper as 123, etc, it should be considered, e.g., as 123 appropriately extended to some fixed physical display size.

1.1 The approach
We introduce a clear declarative notation for defining, analyzing and implementing keyed data entry UIs. The approach is based
on regular expressions (regex), which in themselves are not novel: the technical foundations were well-described in a 1980s textbook [1] and much earlier in the research literature. The idea might be compared to a “dependable user interface specialised Lex”—Lex being the 1975-vintage non-interactive lexical analyzer-generator [11]. Nevertheless, a diverse range of important benefits follow for quality interaction by using the approach. Additionally, the notation supports “traffic lights,” a new concept (though independently proposed in 2001 [5]) that enhances UIs.

The approach is simple: the syntax of expected data is defined using regex that are compiled into a specification for a push-down automaton that implements the UI. The approach was developed with dependable UIs in mind, for instance where a nurse needs to reliably enter a drug dose into a drug infusion pump, where keying errors and errors in implementation have adverse outcomes for patients. The approach allows for a clear, rigorous specification, its analysis, and a more dependable user interface. The approach separates syntax from semantics, and therefore permits a simple implementation of semantics as no syntax errors need be interleaved with the semantic analysis. A variety of optional features can be specified, such as secrecy for password entry. Potential feature interaction is reported.

In some ways this is an obvious approach; however, it appears that dependable data entry is not commonly seen as a serious problem and therefore it is implemented in arbitrary ways that are not closely scrutinised. Most data entry systems work perfectly well when there are no data entry errors! The result is inconsistent and unreliable user interfaces when errors do occur. Indeed, it is common for interactive devices, even in safety critical applications, to handle data entry in a variety of inconsistent and even sloppy ways [21]—possibly exacerbated because different parts of the UI are implemented by people who do not communicate well-enough with each other. The underlying problem is that there is no clear specification for data entry, and in many cases it is easier to repeatedly “reinvent the wheel” in a general-purpose programming language than be consistent. The result in the worst case is that every data entry field has its own handler, working in an idiosyncratic way—often handling user error in different ways inconsistently.

Number entry, whilst apparently trivial, is handled very badly by many user interfaces, not just web-based forms [21]—from office applications to medical devices. User errors, such as the user keying additional decimal points, often go undetected or cause numerical errors. Detecting and blocking user errors can halve the rate of “out by ten” errors, where the number accepted by the device is at least ten times out from the intended number [21]. Other problems in data entry are also possible, such as creating mode errors. Data entry problems appear almost everywhere, not just with number entry, and in a wide range of domains: spreadsheets, calculators, medical devices and so forth [21].

This paper presents a tool-based approach and its principles that, if refined and adopted, could help avoid such problems. Moreover, we will argue that the approach naturally leads into an improvement for users to help them key data more reliably and be more aware of and hence better able to manage their keying errors.

2. DEPENDABLE KEYED DATA ENTRY

As a user enters data, the keys they press are appended to a buffer. The buffer contents are rendered and made visible to the user in a display (except in some password or secure applications, when the display may be masked), and on completion of the data entry, an application processes the buffer to obtain a value. The application gets or processes the value when the user enters the data, which is often achieved by pressing a special key, enter, say, or by continuously providing audible feedback; moreover, if a user presses a key and it does nothing, as a feature of our design, then there is no click.

In this paper we will take the meanings of these italics terms above to be self-evident, though it must be noted that what a user sees (the contents of the display), the buffer contents, what the user keyed, and the value processed may all be different. For example: the display may not be big enough to show all of the buffer contents; the value the application processes may round or otherwise ignore some parts of the buffer (say, treating 1.99 as 2); if the buffer is full, pressing keys may provide no key clicks.

As a user enters data, we provide clear feedback of correct data entry (security systems are an exception where we do not want anybody else to possibly see what or even how much the user is keying). If a user makes a keying slip, we ensure this error is clear to the user, and block the entry of an invalid buffer to the underlying application. Always, undo removes the last character typed, and this always works: the user interface never ignores the user’s keystrokes, even if the buffer has overflowed.

There are many further considerations: if a user starts entering a password and walks away, the system may time-out and reset the partially-entered password; on some forms, entering a field updates other fields the user can see (e.g., a currency conversion form may display two currencies simultaneously, and either updates when the user edits the other currency).

Many UIs are run in an organizational context where the user is made responsible for errors; thus if you buy a higher-priced ticket for travel on the wrong day, then the business on the other end of the web site will treat this as a legitimate transaction, and its rules for refunds will not be sympathetic to “user error.” In other contexts, errors may result in adverse events, worse than wrong tickets, such as over-dosing patients with drugs, or flying aircraft into the ground. The possible problems are very real: for example if a user thinks they entered 2.3 but the user interface responds as if they entered 23, there may be significant problems.

Many new standards attempt to address some of these issues. For example the World Wide Web Consortium’s HTML 5 specification defines text entry fields with data validation to limit values to whole numbers, floating point values, dates, and so on. The NHS has defined a “common user interface” (CUI) that defines many low-level features such as date entry. If nothing else, these standards help ensure a consistent user interface across different applications—provided the low level details are defined, as this paper does.

2.1 Previous work and problems addressed

Data entry validation is generally driven by the desire to provide as rich and responsive an interactive user experience as possible while trapping data entry errors (and enabling users to manage them) before passing data over to a server. PowerForms [5] (devised in 2001) use regex and a constraint language to specify forms for web applications. The regex are written in HTML or Perl, and compiled to specify acceptable field contents. Fields can be marked with status icons, typically traffic lights, much as we propose. PowerForms allows forms to be dynamic: for example, only if the user is married is the field for entering the spouse name shown. See [22] for a review of web data entry validation techniques.

Topes [15] recognize categories of string data (such as email addresses) and ensure that user interfaces process data more reliably; topes provide a flexible scheme intended for end-user programmers, such as people defining spreadsheets for others to enter data into. They do not address low-level interaction issues that we
are concerned with in this paper, and in many ways topes and the approach suggested here are complementary.

We are primarily concerned with providing a high quality, consistent UI that blocks and helps manage errors. Unlike web-based approaches, we are concerned with small details such as exactly how a keystroke modifies a field—this is an issue most approaches leave up to the browser (with variations due to different versions and different platforms). For example, PowerForms provides no easy way to specify field size limits or time-outs. The client/server or browser/server distinction is not relevant to us, but the precise logic presented to the user is paramount. The Appendix provides a list of typical problems.

What is remarkable is that these problems persist, from basic interactive devices with control panels, to web forms (where technologies like PowerForms [5] are available). Perhaps the problems seem so trivial that they are not noticed, and due care is not taken? A very clear, general and easy solution is essential, and it must address the key problems above in all their forms. Furthermore, it has to be sufficiently flexible so that developers find it meets their various needs, yet sufficiently uniform to reduce user transfer errors.

Another possibility is that some or all of the problems are unavoidable in principle; if so, ad hoc programming around them cannot improve matters.

2.2 Predictability & divergence

A dependable UI should be predictable, in the sense that what the user correctly intends to achieve with the interface should be the outcome. Unfortunately users make slips, so a dependable UI must manage errors somehow, and in particular the management of error should itself be predictable.

Predictability can be divided into two forms we will call input predictability, and output predictability. A user has in mind some goal, and they then generate input to the system. For this to be successful, the system must be input predictable. Thus if a user keys [T 2 J 3] DELETE the input would be as if they had keyed [T 2]. For output predictability, when a user looks at a system they can work out what to do to achieve their goals. Thus if the display shows [12] pressing [DELETE] may be supposed to change it to [1]. Many devices implement predictability in different ways, so how a particular device works may be different than in these examples. Moreover many devices implement input and output predictability in different ways; we say their UIs diverge. A common form of divergence occurs when a user keys more than will fit in the display. We say that a system breaks down when predictability breaks down in the absence of divergence; an example would be that a device only records one decimal point, so [5] [•] [DELETE] [5] is treated as [5.] not as [5.0]. A more subtle form a divergence occurs when a display fails to distinguish certain letters and digits (because of poor font choices).

The notion of user itself is problematic. From the system’s point of view, user predictability is that user’s behavior is consistent with an as-yet unknown goal of the user. User predictability breaks down when the user changes for another, or when a user fails to complete a sequence of actions to the point of achieving a goal. Systems often attempt to maintain user consistency by time-outs. Divergence then occurs when the system considers the user has timed-out (and thus may be a new user with a new goal) but the original user with their unchanged goal is unaware of the time-out. In dependable systems, user predictability may require password security, itself a problematic feature, not least because it may be unreasonable to maintain output predictability as this could disclose a password to a bystander.

Conversely to user predictability, system predictability refers to whether the device or system is predictable. System predictability is compromised by variation in designs between devices that look similar (including differences caused by software or firmware upgrades), variety of designs in an organisation (e.g., hospitals with many varieties of infusion devices), poor software engineering practices leading to nobody knowing what the devices really do (which in turn reduces the value of training and user manuals).

In general, UI divergence is unavoidable, although it may be mitigated with dedicated devices and unlimited memory and display capacity (and hence scrolling). For control panel devices (including form fields in GUIs), there are unavoidable boundary cases that cause divergence, and ironically they arise most commonly as a consequence of user error or user/device model inconsistencies. A case in point [21] is a user keyed [T 3 J 5] but the device, finding the input exceeded 99, ignored the decimal point and displayed and acted on [100].

Divergence and breakdown can be detected by the system, for instance when a user keys more than can be displayed. We assume that a dependable system must alert the user— or someone—(perhaps by sound) of a divergence or break down to enable timely and appropriate action. Of course, in some situations, alerts may counter-productively escalate the error rate. Standard examples include control panels with multiple UIs where a user may be unable to prioritize multiple alerts; where alerting a user to a break down may increase their stress and hence increase their error rate; and, indirectly, in cases such as in hospitals where alerting a nurse to an error may encourage a patient to avoid that nurse (and hence appropriate treatment) because of the nurse’s apparent propensity for error. It is helpful to introduce the term alert predictable to indicate when alerts have predictable, intended, effects.

In summary, dependable systems should be predictable, but predictability always eventually diverges or breaks down (because of the certainty of eventual user slips and the possibility of design or system defects); in such cases, the systems should be alert predictable, so that users (or systems) can appropriately manage the problems without escalating error.

Implicit, unexplored, ad hoc and diverse solutions to these problems are the norm, and cause further problems, which are often under-reported since there is no reliable mechanism to detect them.

3. TOOL NOTATION

We base our TUI approach on a regular expression (regex) language. Our notation is clear rather than concise; relaxed about layout and white space; sufficiently but not confusingly expressive; use conventional notation and concepts where possible; and, as regex can be confusing, to permit comments. Crucially, there is a clear distinction between syntax and semantics, and in our approach semantics beyond what a regex can specify is separated out, thus making the specification much clearer.

We introduce the regex notation by re-engineering the HTML 5 "microsyntax" the World Wide Web Consortium (W3C) uses to define “floating point numbers” for user input. (The interactive examples in the Appendix are harder to follow than the non-interactive W3C specification used as an example here.) The W3C should be in a position to use best practice, but their approach is ad hoc and arbitrary. They repeatedly re-implement very similar parsing strategies; they interleave syntax, semantics and error detection in obscure ways. Their notation (English) does not help. They use words, including “fail” and “abort,” in undefined ways.

The W3C specification for floating point numbers is as follows [23], though the original interleaves English “semantics” to determine the value (or overflow) of the number parsed and it also refers to characters using Unicode values.
1. Let position be a pointer into the string being parsed, initially pointing at the start.
2. Skip whitespace.
3. If position is past the end of input, return an error.
4. If the character indicated by position is a `:`, advance position to the next character. If position is past the end of input, return an error.
5. If the character indicated by position is not one of 0–9, then return an error.
6. Collect a sequence of characters 0–9.
7. If position is past the end of input, finish.
8. If the character indicated by position is a `:`, run these substeps:
   a) Advance position to the next character.
   b) If position is past the end of input, or if the character indicated by position is not one of 0–9, then finish.
   c) Advance position to the next character.
   d) If position is past the end of input, then finish.
   e) If the character indicated by position is 0–9, jump back to the step labeled fraction loop in these substeps.
9. If the character indicated by position is an `e` or `E`, run these substeps:
   a) Advance position to the next character.
   b) If position is past the end of input, then finish.
   c) If the character indicated by position is a `:`.
      i) Advance position to the next character.
      ii) If position is past the end of input, then finish.
   Otherwise, if the character indicated by position is a `+`.
      i) Advance position to the next character.
      ii) If position is past the end of input, then finish.
   d) If the character indicated by position is not one of 0–9, then finish.
   e) Collect a sequence of characters in 0–9.

Translated into a regex in our regex notation, the W3C specification becomes:

```
1: whitespace* (:- (?: skip any whitespace
2: \[ minus \]
3: digit+
4: \{ dot digit+ \}
5: \{ exponent-symbol \}
6: \[ (-?: number can end after exponent! \]
7: \[ plus | minus \]
8: digit+
9: \]
10: \}
11: \} 
```

This is notably shorter and easier to read, and less ambiguous as it does not rely on ad hoc phrasings—in fact, it does not permit any. Line numbering is provided by the online demonstrator.

The regex language allows expressions to be of limited lengths, however the W3C specification does not do this: the W3C specification does not worry about overflow, and would accept an arbitrarily long input such as 100000... and return a invalid value and report no error (the incorrect W3C code is not shown here).

In most regex notations, there are two sorts of character: literal symbols and operators. This causes many readability problems, for example if you want to recognize a symbol that happens to be an operator then the operator has to be escaped. Comments in conventional regex are impossible—which is ironic given how unreadable they can be! (Perl has regex comments, but then the comment symbol # itself has to be escaped to be used; we avoid this problem too.) Our notation is like a conventional programming language: there are four sorts of symbols: variables, operators, literals, and comments. White space (outside of literal strings) is ignored.

Line 1 uses whitespace, a variable defined elsewhere, to represent blank text. The name whitespace is followed by the operator *, which means “match zero or more expressions” (the conventional Kleene star). The final part of line 1 is comment, everything including and following (\ - to the end of the line)—chosen as it allows emotionally-annotated comments. In summary, line 1 corresponds to W3C’s English, “Skip white space.”

Line 2 uses the brackets [ ] to make minus optional. This line corresponds to W3C’s English, “If the character indicated by position is a ‘:’, advance position to the next character. If position is past the end of input, return an error.” We show below how such input errors are managed automatically using “traffic lights.”

Two further operators are used in the example here. Line 3 shows the + operator, which is similar to * but matches one or more expressions. Thus, re* = [re+] and re+ = re re*.

Line 7 shows alternation, represented by the | operator: [plus | minus] will match nothing, match plus, or match minus (however plus and minus are defined).

Definitions such as exponent-symbol = “e”| “E” can be placed anywhere, before or after use of the names. A definition uses regex and can itself rely on other names. Since the language defined is regular, the only real restrictions are that names must be defined but cannot be defined in terms of themselves. Naturally, the system reports such errors, as well as the less obvious errors of redefining names or defining names that are not used anywhere (or are only used in definitions that themselves are not used).

Strings represents characters concatenated; thus "abc" is the same as writing "a" "b" "c", etc. The empty string is not permitted.

So, what have we achieved?

- Brevity and clarity.
- The language requires some things to be explicit that are only implicit in the W3C English: we have to explicitly say that a number can be followed by anything (line 11), whereas the W3C implicitly accepts nonsense like 2.2.4.
- If we wanted a better definition, there should be no line 11: we should forbid anything (other than blanks, perhaps) following a number (surely?) as part of a number.
- Lines 6–9 show that a number can end with E, as in 2.3E, the W3C does not require a number with E to have an exponent. As it happens, it treats 2.3E as 2.3E1.
- On the other hand, as lines 7 and 8 make clear, if the E is followed by + or – it must be followed by a digit. This is clearer in our language than in the W3C specification.
- The language compiles into a push-down finite state machine, which can run the user interface specified. The W3C has to be converted (unreliably!) by hand.

### 3.1 Display and buffer capacity

Crucial to the approach is handling all forms of user input including errors, and handling data exceeding the input buffer length is an important design consideration.

A variable value-display-size sets the size of the display, and hence the maximum size of the buffer (the buffer grows as the user presses keys). The regex is checked to confirm that the longest shortest (i.e., of all shortest matches, what is the longest?) matching input is permitted. This checks that the regex has not specified input that is unavoidably longer than the display. It is still possible for a user to enter too much, either incorrectly or correctly—for instance "X"* will match any number of [X] keys. Even with a compile-time check, the buffer length must still be monitored while the user is entering data.

There are interesting cases. Suppose the specification is digit+ . " digit+ and the display is set to 10 characters. If the user keys digits 9 times, then the user cannot enter the required dot digit+ as the display is not big enough. We do not report this situation as it might be confusing that the device reports an error, yet the user cannot see one! Instead, an error is reported, as usual,
when a key is pressed after the display is full; this is an unequivocal error. Possibly, the use of * + and rather than explicit bounds on repetition is one part of this problem.

When the user’s input exceeds the appropriate defined length, the TUI behaves as if it is in a red state (see below) and the display is modified to alert the user that overflow has occurred—currently $\prod$ is displayed (which also necessarily replaces some of the display positions). When this happens, the TUI need only count user keystrokes, to ensure that $\text{UNDO}$ can be implemented properly (on a GUI-based version, all keystrokes would need to be retained in case the user moved the cursor). Crucially, a user operating the device with “their eyes shut” (i.e., not paying full attention) will always find that $n$ key presses followed by $\text{UNDO}$ is equivalent to $n – 1$ key presses, with the last one deleted—even if the user cannot see the last keys in the display.

3.2 A more sensible number definition?

After criticising the W3C floating point definition because of its inscrutability and weak error management, here we propose a better specification, not least because it is only 2 lines rather than 30!

```
blank*   [sign] digit+ [dot digit+]  
[exponent-symbol [sign] digit+]  blank*
```

We assume that names are appropriately defined elsewhere. In contrast to W3C, our treatment of signs is consistent for the mantissa and exponent; an exponent is required if the exponent symbol is present; blanks are ignored both before and after a number; and the number is not allowed to finish with “anything” and therefore cannot finish, confusingly, with more numbers—this avoids the W3C problem of permitting data like 2.3E.4, 1EE8 and 2.3.4.

4. EXAMPLES

An advantage of using a state machine is that states can be coloured automatically—and thus consistently—to give the user feedback on their progress in entering data or for signalling errors, as the case may be. The notion of coloring derives from [21].

If the buffer is syntactically incorrect, then the color will be red; if the buffer is correct, it will be blue; if the buffer is a prefix of a correct value, it will be yellow (amber). Finally, if the buffer is equal to a value previously accepted by the application, it will be green. State colors are also represented to the user with sound and icons (see figure 1).

If the user’s input exceeds any length constraints, it is colored red. (The length constraints are propagated back; the user may therefore be warned about a length constraint before the display is full when the only possible ways of completing the input from the current state would exceed the length.)

Optionally, the buffer can be colored by the application on every keypress, as we hinted above to handle cases such as the varying number of days in February. However, the application is only allowed to make the color “redder” as it is not allowed to correct errors the user has made.

The Institute of Safe Medication Practice (ISMP) defines safer ways of writing numbers [21] to avoid potential confusions such as 150 and 150.1 leading to patients getting drug doses that are misread—in this case, by a factor of ten. The ISMP defines its rules in English (which, unlike the W3C, is appropriate for their readers, who will include non-technical clinicians writing drug doses by hand); see [21] for more details. The regex specification for the ISMP rules is shown in figure 4. This compiles into the FSM shown in figure 5, generates the JSON shown in figure 7, and generates the TUI illustrated in figure 8. (The TUI obviously provides a richer experience than the pictures can convey.)

Figure 6 visualises the ISMP specification. It is helpful [4] and not difficult to generate visualizations of regex; the visualization helps developers literally see their specifications more clearly, and hence help detect typos or other mistakes or misunderstandings in them, and since the visualization is generated from the FSM rather than the parse tree [6], it goes beyond merely re-presenting the regex: it represents (in the visual notation) what the specification means. The slight disadvantage of this approach is that the visualization is not necessarily formatted for clarity for end-users rather than developers. See [4] for more ideas.
4. Time-outs and interruptions

Time-outs and interruptions can be problematic if the regex does not support any autocompletion possibilities (i.e., in every state the user has a choice, so no autocompletion is possible). Another problematic combination is mask together with key-lighting, since highlighting compromises the secrecy of masking. If a user was entering a password, autocompletion enabled would mean the user would not even need to know any key of the password! In fact, in the special case that no input is ever required from the user, the compiler complains.

For any specification, a summary of features and feature interaction is provided. For example features: mask complete generates a report warning that these two features are (almost certainly) inconsistent. Autocompletion alone is problematic if the regex does not support any autocompletion possibilities (i.e., in every state the user has a choice, so no autocompletion is possible). Another problematic combination is mask together with key-lighting, since highlighting compromises the secrecy of masking. If a user was entering a password, autocompletion enabled would mean the user would not even need to know any key of the password! In fact, in the special case that no input is ever required from the user, the compiler complains.

5. Optional Features

The TUI generated is consistent and carefully designed to promote dependable behavior. In particular, every non-operator (text, digit, etc) key pressed is added to the input buffer (pushed on to the PDA's stack), and the display changes only:

- by the addition of keys pressed by the user, which are appended on the right of the display. More than one symbol may be added to the display if autocompletion is enabled.
- by the user pressing **UND** (delete the last character from the buffer).
- by the user pressing **CLE** (clear the buffer).
- by the user pressing **RES** (the last confirmed value, if any, replaces the buffer).
- once a key is rendered in the display, it does not change.

Since developers will want variations depending on the context of use, the regex specifying the basic behavior of the TUI is supplemented with a separate “feature list” to specify various optional features, such as whether the display should be masked (for password entry), whether there is an undo key, and so on. Relative to other approaches to UI design, this approach is unusual: the features chosen for the UI are explicit, rather than embedded in (and probably distributed around) code—perhaps implemented differently in different parts of the UI (as frequently happens [21]).

In conventional regex, the syntax may be `/pattern/modifiers`, where optional modifiers are a list of letters like i to ignore upper/lower case distinctions, g for global matching, and so on (and often the regex is compiled to an object itself with further properties). In our approach features (“modifiers”) are only introduced in the specification by a list of mnemonic names (and optional values), for example: no-key-undo mask key-lowercase defines a TUI with no undo key, masked data, and distinguishing upper and lower case keys (so it probably has a shift key). The general idea is that the default feature is “best,” and only variations need to be specified. Thus the property for the undo key is not that there is a key for undo, but that requesting no undo key has to be explicit.

Some features can be assigned values. For example, if it is preferred to call the undo key Delete, then name-undo-key = "Delete" reassigns it.

When a specification is compiled, a summary of features and feature interaction is provided. For example features: mask autocomplete generates a report warning that these two features are (almost certainly) inconsistent. Autocompletion alone is problematic if the regex does not support any autocompletion possibilities (i.e., in every state the user has a choice, so no autocompletion is possible). Another problematic combination is mask together with key-lighting, since highlighting compromises the secrecy of masking. If a user was entering a password, autocompletion enabled would mean the user would not even need to know any key of the password! In fact, in the special case that no input is ever required from the user, the compiler complains.

For any specification, a summary of feature settings as well as any conflicts and any potential weaknesses (e.g., switching off key clock sound) is generated by the compiler.

5.1 Time-outs and Interruptions

Inevitably, users will sometimes be interrupted when entering data. The most common approach to handling interruptions is to time-out the user interface in some way, but perhaps subtly in a way that a user may not notice [19]. In our approach, a flashing time-out icon (figure 1) is shown, and its rate of flashing increases
Figure 6: A “railroad” syntax diagram automatically generated from and visualizing the ISMP definition from figure 4. Railroad diagrams are more readable and more suitable for users than transition diagrams.

```json
{name: "ISMP numbers",
 alphabet: ".0123456789",
 autocomplete: false, // autocomplete-any
 ...}
states:
{error: false, accept: false,
 arcs: [5,1,2,2,2,2,2,2,2,2], auto: null},
{error: false, accept: true,
 arcs: [3,5,5,5,5,5,5,5,5,5,5], auto: null},
...}
```

Figure 7: Extract of the JSON generated from figure 4. Although identical in this example, the device keys and the FSM alphabet may be different if autocompletion is enabled; other details are beyond the scope of this paper to discuss.

(Along with ticking sounds) as the time-out approaches. Thus the user is given a warning (perhaps 10 seconds) within which they can respond and reset the time-out. If the time-out happens, the display is reset, so that a new user does not continue by accidentally modifying the current data.

This approach contrasts with some commercial devices where a time-out occurs silently and are not visible until the user hits another key [19]. Although there are many other ways of handling time-outs, the approach here appears sufficiently general. The feature system also allows the TUI to have no time-out at all.

6. TECHNICAL ISSUES

A rule of the TUI design is that key presses are appended to the buffer and rendered for the user to see. Furthermore, once rendered, key representations do not change. There are three issues:

- The buffer may be full, and it is not possible to render the key at all.
- Some keys may need to be rendered differently. To improve readability it is recommended to render digits after a decimal point as smaller, thus: 
- The key legend and the display representation may need to be different. For example, the space key is generally blank, but the display might be better represent it as , as an explicit blank; or one might wish to render the displayed decimal point larger than conventionally, etc.

The tool warns if white space is specified in the key set, since in general it is hard to see in a display (and it might be accidentally typed into a regex). Unfortunately, warning about visual ambiguities is fraught with difficulties. However, the tool warns if S is confusible with 5, O with 0, or I (letter) with 1 (digit) or L, as this may matter and be unavoidable with certain fonts.

The regex are compiled in the normal way into a deterministic FSM. States are then colored: the final state is blue and all other states are colored orange (amber). Some states will not have outgoing edges for all symbols in the alphabet (i.e., when there are no transitions specified for some keys a user could press): if so, a single new state colored red is added and transitions from states for all otherwise-undefined keys are added.

If a state has a unique non-error transition, then this will be flagged for autocompletion (if autocompletion is enabled), though if autocompletion-to-end is enabled, the non-error transition will only be flagged if a path of unique transitions leads to an accepting (green) state.

A user keys text and operators. Text are the symbols that make up the user’s data, such as digits or words, and operators are ways to change or use the text, such as \[\text{undo}\], \[\text{confirm}\], etc. (Not all operators need be provided in any specific TUI.)

When a user keys text, the text is appended to the buffer, and the FSM makes a transition. If the user keys \[\text{undo}\], the last text symbol in the buffer (if any) is deleted; if the user keys \[\text{confirm}\] all text is cleared from the buffer. If the user keys \[\text{confirm}\] when the FSM is in a blue state, then the application is given the buffer.

The operators \[\text{undo}\] etc. operate on the buffer not on the display. They therefore edit exactly what the user has keyed (if, for example, autocompletion is enabled, what the user can see need not be what they keyed). This approach supports input predictability.

The buffer is a stack; together with initialisation of its contents, the only operations are: clear by \[\text{cancel}\], pop by \[\text{undo}\] and push current key. Hence the buffer and the FSM comprise a PDA.

The feature value-display-size sets the maximum (physical) display size. The accepting states within the regex are annotated with the constraint and a unique ID; the constraint is then back-propagated through the states. (It is possible that length constraints will impact autocompletion—it is obviously inappropriate for an autocompletion to overflow the display!) During input, if the buffer exceeds the length constraint, the current state is treated as if colored red.

Finally, if autocompletion is enabled, then each state has an “auto transition” which, if enabled, specifies a key to insert in the dis-
play (the target state of the transition necessarily specifies a unique keystroke). The keystroke is not inserted in the input buffer—thus autocompletion is transparent to UNDO.

7. DESIGN ISSUES

Keyed data entry is ubiquitous and therefore used for very many purposes, not all equally appropriate.

If an application requires very complex data formats, that may be a source of user error regardless of details of the design of the user interface. Our approach does not simplify the application semantics; its main achievement is to simplify and clarify the syntax used to specify data input.

The domain where the user interface is used is obviously a very important factor in human error and system-induced error. A diver wearing gloves and perhaps with cognitive defects induced by their extreme environment may need a very different style of user interface than a nurse doing a busy ward round; a person sitting at home relaxing with a hot cup of coffee paying for purchases off the web; a school child using a calculator in an exam; a person asking for cash from a cash machine; and a crook trying to break in to a security system. These are all examples of keyed data entry applications! There are no grounds for assuming that a uniform style of interaction (e.g., using a web-conformant technology) will be ideal in all circumstances.

A common method to reduce data entry errors is to introduce check sums or check digits. (ISBN numbers are a familiar example.) Many approaches such as this, which go beyond the syntactic issues of keyed data entry, can reduce undetected data entry errors.

On the other hand, some forms of data entry may be better implemented with different styles of user interface. Choosing the day of the week, for example, might be better thought of as a menu selection process than as a keyed data entry problem. Picking from a visible list may be more reliable than keying text, for example. A menu selection interaction technique, suggested in 1978 [16, 17], which is ideal for such applications (specifically, being key-based and predictable) cannot be simulated with the approach described in the present paper. Again, techniques optimized for small keypads [12] with multiple meanings per key (e.g., [ABC] on a single key), common on mobile phones, also use techniques that the present approach may not be able to support. There is scope for empirical work to inform design decisions.

Imprecise numbers might be more reliably entered using sliders or by knobs than entered using discrete numeric key presses. There are complex domain trade-offs that are beyond the scope of this paper: people can use familiar keypads without looking at them, whereas a knob must be looked at to see where it is set.

The production costs of different styles of UI may be an important driver in design decisions. While this may not be a significant factor for web-based user interfaces (the user has already paid for the PC, the physical UI), for control panels, reducing button counts—whether for cost or space reasons—may be a significant concern. Up/down keys may significantly reduce the number of keys to manufacture (perhaps from 11 to 2 or 4) and thus be almost irresistible for manufacturing purposes, yet at the same time such a change in a UI raises trade-offs in the user’s performance and error rate that for the application domain may be more significant than the up front cost-saving. Accidentally hitting $\mathcal{A}$ may change a value, perhaps by a factor of ten, whereas accidentally hitting the decimal point alone—if detected—is not a valid value.

8. EMPIRICAL ISSUES

Much evaluation in HCI involves empirically evaluating users—behavior, performance, or qualitatively—whereas comparatively lit-
form $A/B$ tests. A simple question: are displays better left aligned or right aligned? The current approach allows a developer to specify either choice, and makes controlled experiments much easier, since it guarantees that it can generate exactly equivalent user interfaces differing only in the controlled properties.

With left alignment, the change to the display when a user presses a key is always the key that the user pressed. With right alignment, every key press causes the entire display to change, in fact, to shift left. In the case of displaying 111 if the user next keys $\bar{1}$, the display appears to put a 1 on the left of the 111 as only that part of the display changes, but in fact it was appended on the right. Thus if the user keys $\bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1}$ while looking at the display, they may expect it to show $\bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1} \bar{1}$ since it appears to be building up from the left as they key! In contrast, left alignment supports the principle of display inertia [7] and appears to raise no predictability problems—it always appears to build from the right, as in fact it does. Nevertheless, we do not know which is better or under what circumstances, even for entering conventional left-to-right Arabic numerals.

The approach supports analytic evaluation; it generates a report of feature interaction (see §5) and generates a formal specification of the FSM (e.g., in JSON). Further formal analysis can include building Markov models [18], but drawing the graph of a state machine is often sufficient. For example, for "{\text{\texttt{jan}}} "{\text{\texttt{uary}}} | \ldots | "{\text{\texttt{dec}}} "{\text{\texttt{ember}}}", visual inspection of the graph would be sufficient to see that all months are covered, and that for each month a 3 letter abbreviation is permitted.

Finally, how developers use the system to define more dependable UIs can be evaluated and hence lead to an improved approach. We do not know what mistakes or misunderstandings developers may have. If there is data available on user keystroke sequences, then techniques such as [13], intended to help word processor users detect errors in specifications based on documents they are searching, may be used to help the developer detect outlier errors in their specifications.

9. CONCLUSIONS

We have known that user interface design involves trade-offs, but this paper has shown definitively that even a very simple, confined domain has irreconcilable trade-offs. There are no easy solutions to dependable keyed input, and designs have to consider user error balanced against the goals of the tasks the design is intended to be relied on to support. The corollary is that when explicit, formal trade-offs are not made—that is, almost all the time—user interfaces will (eventually) induce adverse incidents as a result of heedless mishandling of error.

Given the importance of dependable keyed data entry, this paper exposes the lack of empirical data and evidence-based theories to help designers and developers. Seeing clearly that so little is known about design trade-offs in this area is a contribution of this paper. Simple-looking user interfaces look simple, and therefore may be confused with designs that are actually simple (and perhaps reduce errors). Some of the issues raised in this paper highlight how misleading "simple" may be when it comes to interaction; interaction is something that is not visible to the user—and unfortunately may be almost as invisible to the designer.

Since data entry is commonly implemented by ad hoc and sloppy solutions, a diverse range of dependability problems ensue. The problems are to do with data validation, in this case of data input by users. In the UK, data validation is covered by the syllabus of the General Certificate of Secondary Education (GCSE, and the international IGCSE) examinations on ICT. It appears that we are employing professional developers who are unaware of the relevance of elementary procedures normally examined at age 16.

The TUI presented in this paper together provide an unusually convenient and consistent UI for dependable keyed data entry. The traffic light scheme is optional, but provides a consistent way of helping users manage errors. Our solution ensures:

1. Valid data items are specified with a clear and concise notation using regex. The language does not use meta-characters, has comments, and is not sensitive to white space—even though this is unconventional for regex.

2. The data entry specification is a readable, concise textual program, serving as a form of communication between designers, developers and users, and thus helping ensure consistency between implementations.

3. The user’s progress in entering data is highlighted with a distinctive traffic light scheme. The traffic lights remove ambiguities that arise in conventional user interfaces.

4. Because of the approach’s simplicity, explicitness and rapid generation of UIs, it supports effective iterative design.

5. The specification compiles into a PDA, which is efficient and correct by construction $(C \times C)$ [10]. The PDA can be analysed formally.

6. The compiler generates a JSON specification of the user interface (see http://www.json.org; [20]), which can be used in almost any programming language, and therefore provides a route for easy implementation of consistent user interfaces in any environment.

7. Data entry properties are specified with named features, providing tailored support for autocompletion, time-outs, masking, key highlighting, whether there is $(\text{UNDO})$, and so on. Feature interactions are reported.

8. All relevant design issues are clear and explicit.

We note that FSMs and PDAs are simple, well understood and easy to analyse. They can also be compiled into error-free code (or into hardware) that will not have run-time errors. Following Peter Ladkin, we assert that if it is possible to use a FSM, then there are overwhelming reasons to use one. FSMs are efficient: they are as fast as one can make any system, and they can be compiled to simple hardware to build user interface controllers.

Although compiling the language is fast, it is simple to separate compiling from running, and pre-compiled PDAs can then be passed to an API (currently using JSON) to implement different user interfaces as desired. Typical PDAs generated are small, around ten states, and many are smaller.

A prototype implementation at harold.dimbbleby.net/regex. The examples in this paper work on that system.


10. REFERENCES


arises because the user cannot tell whether display change to a decimal point). When the user keys a digit, say for sake of brevity, that our approach sets out to avoid or solve:

Here are some issues, restricting our examples to number entry for

EXAMPLE NUMBER ENTRY ISSUES

- Many numeric displays initially show (zero followed by a decimal point). When a key is pressed, say for the sake of brevity, that our approach sets out to avoid or solve:

- Many devices, particularly those using seven-segment displays, show the decimal point as a very small dot that is inconspicuous. Digit spacing is often unaffected by the presence of a decimal point, again making it hard to see.

- Many calculators get 45% rather than 4.5% because the user had keyed (consider calculating the percentage of the world population from the US. Many systems handle input with code that is rewritten for each device."

2. More generally, many UIs display a default value that is replaced by the user’s first keystrokes. Thus although the display may show , pressing [DELETE] has no effect, and certainly does not change it to —because the is a default and nothing has been keyed by the user to delete.

3. When a user accidentally presses two decimal points almost all UIs display a single decimal point. What does [DELETE] do now? If the user pressed two decimal points then a single delete (to delete the second) would leave the display with no decimal points.

4. Many systems handle input with code that is rewritten for each input field, and therefore handles input inconsistently [21].

5. Many devices, particularly those using seven-segment displays, show the decimal point as a very small dot that is inconspicuous. Digit spacing is often unaffected by the presence of a decimal point, again making it hard to see.

8. Some devices silently limit how much can be entered [21]. Consider calculating the percentage of the world population from the US. Many calculators get 45% rather than 4.5% because the user had keyed (consider calculating the percentage of the world population from the US."

9. Many organisations have a wide variety of devices, each with their own proprietary UI, compound problems as users move from one device to another, inducing transfer errors.

10. When a key is pressed, many devices provide a key-click. It is a potential problem when a key is pressed and provides audible feedback but in fact does nothing. An example is when the user interface has a limit on the number of characters it can handle, and any excess are ignored—but they still key-click, potentially confusing the user. (Keys usually make a single sort of non-discriminating click, and thus confirming that they have been pressed, and that, secondly, they had an effect, are confused.)

11. Devices rarely log user actions and timings and thus neither help in research (how to make better UIs) nor in investigating adverse incidents, nor are they very helpful for organisations or individuals wishing to improve processes. For example, if a device records 55 mg per hour was the drug delivery rate, that does not mean that the operator entered data to request that rate [21]. A summary log from a typical system does not reliably imply the user did any recorded steps.

12. There are specification languages that attempt to solve some of the above problems, however none define how the UI works exactly: they only specify the syntax of input strings. Thus how works, what happens with input buffer overflow, and so on are undefined and therefore may be implemented arbitrarily. As ISO standard 14977 [9] says there is a large diversity of metalanguages for specifying languages and these are often badly or incompletely designed.

Appendix

EXAMPLE NUMBER ENTRY ISSUES

Here are some issues, restricting our examples to number entry for the sake of brevity, that our approach sets out to avoid or solve:

1. Many numeric displays initially show . (zero followed by a decimal point). When the user keys a digit, say does the display change to or to ? This ambiguity arises because the user cannot tell whether is shown because it is the initial display, or because the user (or another user) recently keyed ., or then keying would have changed the display to , but if the user had previously keyed nothing, if the user keys then the display would change to or to . (The display should only show the user’s data, not default or initial values. Additionally, showing a trailing decimal point should be avoided.)

2. More generally, many UIs display a default value that is replaced by the user’s first keystrokes. Thus although the display may show , pressing [DELETE] has no effect, and certainly does not change it to —because the is a default and nothing has been keyed by the user to delete.

3. When a user accidentally presses two decimal points almost all UIs display a single decimal point. What does [DELETE] do now? If the user pressed two decimal points then a single delete (to delete the second) would leave the display with no decimal points.

4. Many systems handle input with code that is rewritten for each input field, and therefore handles input inconsistently [21].

5. Many devices, particularly those using seven-segment displays, show the decimal point as a very small dot that is inconspicuous. Digit spacing is often unaffected by the presence of a decimal point, again making it hard to see. 7-segment displays are inappropriate in portable devices (e.g., insulin injector pens) that may accidentally be read upside down, perhaps reading 18 as 81.

6. Many devices have time-outs. Keying with a pause may result in entering just without warning. Furthermore, devices handle time-outs differently—for example if a decimal point has been keyed, the decimal part of the number may be zeroed but not the integer part.

7. When devices allow fractional numbers with decimals, the decimals are often shown in the same font and style as the whole number part; recommended practice for readability is to have a larger decimal point and for the decimal part to be smaller. Compare (drawn to scale) with .

8. Some devices silently limit how much can be entered [21]. Consider calculating the percentage of the world population from the US. Many calculators get 45% rather than 4.5% because the population of the world needs 9 digits, the last of which is ignored.

9. Many organisations have a wide variety of devices, each with their own proprietary UI, compounding problems as users move from one device to another, inducing transfer errors.

10. When a key is pressed, many devices provide a key-click. It is a potential problem when a key is pressed and provides audible feedback but in fact does nothing. An example is when the user interface has a limit on the number of characters it can handle, and any excess are ignored—but they still key-click, potentially confusing the user. (Keys usually make a single sort of non-discriminating click, and thus confirming that they have been pressed, and that, secondly, they had an effect, are confused.)

11. Devices rarely log user actions and timings and thus neither help in research (how to make better UIs) nor in investigating adverse incidents, nor are they very helpful for organisations or individuals wishing to improve processes. For example, if a device records 55 mg per hour was the drug delivery rate, that does not mean that the operator entered data to request that rate [21]. A summary log from a typical system does not reliably imply the user did any recorded steps.

12. There are specification languages that attempt to solve some of the above problems, however none define how the UI works exactly: they only specify the syntax of input strings. Thus how works, what happens with input buffer overflow, and so on are undefined and therefore may be implemented arbitrarily. As ISO standard 14977 [9] says there is a large diversity of metalanguages for specifying languages and these are often badly or incompletely designed.