Electronic Communications of the EASST
Volume X (2011)

SUBMITTED TO

Proceedings of the
Fourth International Workshop on Formal Methods
for Interactive Systems
(FMIS 2011)

Dependable keyed data entry for interactive systems

Harold Thimbleby

16 pages

Guest Editors: Judy Bowen, Steve Reeves
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Dependable keyed data entry for interactive systems

Harold Thimbleby

h.thimbleby@swansea.ac.uk

Abstract: Keyed data entry is fundamental and ubiquitous, occurring 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 destinations for GPS navigation, to name but a few of its applications. Unfortunately data entry is often implemented poorly.

We introduce *divergence*, a loss of predictability in a user interface, and show that it is in general unavoidable in data entry, and therefore a systematic approach is called for. This paper presents one such an approach. Many inter-related ideas "fall into place"—e.g., autocompletion, prompting, automatic color coding—through the approach. The approach contrasts with conventional systems that are generally inconsistent and unhelpful to users, particularly after errors.

Keywords: keyed data entry, divergence, FSM, user interface properties, dependability

1 Introduction

What are appropriate formal properties for dependable user interfaces, and how can they be used to help practice? This paper explores the area of dependable keyed data entry: an area in need of formalisation (as we shall see) and in need of tools for correct implementations, but tools sufficiently easy to use and sufficiently flexible for developers that the approach is adopted and hence delivers benefits to users and their work.

The original motivation for the work reported here was the discovery that medical devices, such as drug delivery systems, use *ad hoc* number entry methods, with the result that user interfaces that should be dependable are unpredictable. Some infusion pumps parse numbers in several ways, depending on the mode they are in, and thus provide an unpredictable experience for the user, particularly in the way they handle syntax errors [16]—in medical devices, potentially with the result that patients are given incorrect drug doses. There are examples where devices that claim to log "user actions" have been used to assign blame to users, but where the devices may have logged how they responded to user input, not the direct input itself—the device incorrectly parsed input [17], thus the logged actions were not the actions the user requested.

As section 3 below (and [16]) show, these problems are not limited to specialist devices. The aim of this paper is to show that dependable interactive data entry (the generalisation of number entry) is not a trivial problem, but that a better approach is possible. The approach proposed in this paper involves specifying the data syntax and generating a state machine to parse the user's input. This approach is efficient, rigorous and general, and allows consistent user interfaces to be generated with a variety of well-defined properties as appropriate to the needs of the application.

Because the approach relies on a compiler, various checks on the user interface and design trade-offs can be made, including raising issues of feature interaction, and these checks ensure a more dependable outcome appropriate to the task domain.

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 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.

2 Previous work

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 before passing data over to an application. PowerForms [6], devised in 2001, use regular expressions (henceforth, regex) and a constraint language to specify forms for web applications. 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 [19] for a review of web data entry validation techniques.

The UK National Health Service has defined a "common user interface" (CUI) that defines low-level features such as date entry. Such standards try to ensure a consistent UI across different applications *except* the low level interaction details need to be well-defined and appropriate, and as we will show below they fail to do so. Recent standards work [4] ignores dependability and managing user error *per se*—but is adding user experience (UX) issues like pleasure and comfort. Indeed, ISO standard 14977 [9] says there is a large diversity of meta-languages for specifying languages but that these too are often badly or incompletely designed (as epitomized by the W3C case study in section 4.1).

It is remarkable that these problems persist. Most likely, programmers find it easier to use general-purpose programming languages and then they naturally make *ad hoc* implementations—since data entry is deceptively trivial. Instead, arguably, we need an approach where a dependable UI is constructed automatically from a specification; this is an example of "correct by construction" (C×C) [10].

3 How is keyed data entry problematic?

Many devices have a display and a keyboard, where the user can enter values into the display that are then processed in some way. The most familiar form of this style of UI is found on handheld calculators, but it also appears in form filling, medical devices, cash machines, and other areas.

Best practice is to validate the user's input and then process only correct values. However few

UIs follow best practice, and in any case this is only part of the story. The handheld calculator provides a few examples of the possible problems:

1 ▶ Handheld calculators rarely validate user input. If a user keys [1] [•] [2] [•] [3], probably `1.23` is displayed and processed. Put charitably, calculators *correct* the user input but do so silently—and different calculators correct errors in different ways.

2 ▶ If the user makes a keying error, few calculators provide a [DEL] to delete the last key press.

3 ▶ If the user exceeds the display capacity, typically excess keys are ignored.

4 ▶ When the user has not yet keyed anything, the display generally shows a value (such as `0.`) and this can cause ambiguities and hence problems. For example, if the user keys [1] when the display shows `0.`, the display could change to either of `1.` or `0.1`. Very few calculators display nothing when the user has keyed nothing.

5 ▶ Calculators frequently overload the display: it is used to show the result of a calculation and the number being entered (depending on the mode). In general, the display can show any number (say `42`) and the user keying a digit can either start a new number or extend the displayed number.

6 ▶ Calculators rarely detect overflow in number entry. A typical calculator will perform the following calculation incorrectly: the proportion of the world population that is US is 308,000,000 (the US population) divided by 6,800,000,000 (the world population). The Apple iPhone calculator (v 3.1.3) gives the incorrect result 0.45. The calculator has silently discarded digits the user keyed, because the display only shows 9 digits. However if the iPhone is rotated to portrait, the calculator can handle 16 digits and it will give the right answer. Evidently the iPhone has code in it that *explicitly decides* to discard and ignore excess digits from the user without reporting any error.

7 ▶ Keys on calculators rarely "key click" so that the user can be sure whether (and how often) they have been successfully pressed. A few calculators have key clicks, but the clicks indicate that the keys have been pressed, not that they have had any effect (e.g., as happens when the display is full)!

These issues are very familiar and perhaps easy to dismiss as "obvious." Yet the problems are ubiquitous. Medical devices, such as infusion pumps that are used to deliver controlled amounts of drugs to patients, have very similar problems. Infusion pumps are used by overworked, stressed nurses, who may be interrupted during their operation of a device ... and incorrect values used by the infusion pump may kill patients. Data validation is only part of the problem—for example, how should the device continue to interact with the user after an error is detected? Data validation is often treated as an "validate the data *after* the user has entered it" problem, with no connection to the user's error recovery, data editing, and interaction generally whatsoever.

On PC-based UIs keyed data entry is more sophisticated and some problems can be avoided. However, many problems remain—for example, in many data entry tasks if the user makes a keying error, when they press [OK] (or equivalent) their entire input in the erroneous field is deleted. They may have keyed in their credit card number as `4567 4256 2368 9667` as that is exactly how it is shown on the card, but the system rejects it and clears the user's input so they have to start again—and hence increase the chance of new errors.

The point is that the user experience of keyed data entry seems simple ("*just* key in what you want"), except it conceals many design choices and constraints. The design choices are typically

implemented in many places, and few designers are in full control of them as they affect the user. Worse, in many implementations, many design choices are implicit in the source code structure and interaction between features will be unexplored and undefined (we will give examples later). Implicitness creates dangers for software reuse.

Quite possibly, the iPhone calculator programmer (point 6 above) has just called a standard function `getNumber` and the iPhone operating system itself has made decisions about the number of digits permitted in the field: if so, the calculator code might be impeccable, but is being let down by an over-simple UI widget. And then with agile development methods, the design itself will be a moving target; with a process following ISO 13407:1999 (Human-centred design processes for interactive systems) the design is *supposed* to iterate and improve—and thus a consistent user experience gets even harder to deliver.

The chance that the user experience is consistent, integrated and predictable, especially during error management after a key-slip when consistency is crucial, is low. In addition, as we discuss next, some problems are unavoidable in principle: which means programmers, if they do not realise the problems are intractable, will tend to invent "quasi-solutions" that impose arbitrary boundary conditions on users that, most likely, have not been fully analysed or tested. This may be called cargo cult interaction design [8].

3.1 Predictability & divergence

A dependable UI should be predictable. There are several ways in which a user may expect a UI to be predictable:

8 ▶ The same sequences of keystrokes always result in the same results. In particular, for the keys [DEL] and [CLEAR], for any other key [X], the user may expect [X] [DEL] to have no effect, and for any sequence of keys [...] [U] [V] [W] followed by [CLEAR] to be as if nothing has ever been keyed.

9 ▶ If the display shows something, then [OK] submits it so the device or system acts on the displayed value.

10 ▶ If the display shows some sequence **UVW**, then pressing key [X] (not one of the special keys [DEL] etc) changes the display to **UVWX**.

11 ▶ If the display shows some sequence **UVW**, then pressing [DEL] removes the last character (changing it to **UV**) and pressing [CLEAR] clears it (changing it to).

However, these basic expectations can fail, for example:

12 ▶ If the display is full, say showing **VW**, then pressing [X] may have no effect—since the display is full it remains showing **VW**; then pressing [DEL] may remove the last character from the display, resulting in **V**. Here, [X] [DEL], which might be expected to do nothing, has the effect of [DEL].

13 ▶ If the value shown has a syntax error or is out of range, pressing [OK] may be undefined. Thus pressing [OK] when the display shows **1000** may cause the device to act on 500 (e.g., if this is the largest allowed value). A case in point [16] is a user keyed [1] [3] [0] [•] [1] but the device, finding the input exceeded 99, ignored the decimal point and displayed and acted on **1301**.

14 ▶ If pressing a key would cause a syntax error, the key may not appear in the display or may have some other effect. Thus [1] [•] [•] [3] may appear as **1•3**, and, more interestingly, [1] [•] [2] [•] [3] may appear on different systems (or in different modes) as **1•23**, **12•3** or as **1•3** [14, 16].

15 ► If the display has already overflowed, deleting the last character may or may not reveal an earlier character. Thus pressing [DEL] with the display showing `UVW` (supposing this to be full) may change it to `TUV` or to `UV`.

16 ► Race conditions can affect any form of predictability. Thus, a problem occurs when the user is slow or walks away and is replaced by another user: if the device times out, the display may change on its own, typically as if a user keyed [CLEAR]. A user reading the operator's manual while trying to use the device may read one thing about the device and experience another behavior! Time outs can also create race conditions: the user reads `1` and they wish to change it to `15`: they press [5] [OK] and are surprised when `5` is displayed, not `15`.

17 ► Autocompletion is widely used to reduce user keying effort, however it is normally implemented to confuse the usual relation between display and input. Thus if [J] autocompletes to `JANUARY` then [J] [DEL] would not (normally) change the display to `JANUAR` but to `J`—what it showed before [J] was keyed. Similarly, does keying [J] [K] change the display to `JK` or to `JANUARYK` or is it an error and leaves the display as `JANUARY`?

When a UI fails to be predictable, we say its behavior *diverges* from the user's reasonable predictions. More specifically, we could formalise the laws illustrated above and formalise divergence, though in different ways for different systems. In other words, divergence is a property of the behavior of a UI and the laws of behavior the user has been told (e.g., from documentation). Clearly for a dependable UI, we want to reduce divergence and ensure that the user is made aware of it. **In general, UI divergence is unavoidable even by good design**; it also happens because users make slips or are unaware of the limitations of the design.

We propose that a dependable UI should:

18 ► Define its laws of normal behavior. For example:

- Each key pressed provides audible feedback (key click): one press, one click, and a distinctive key click (e.g., a warning beep) if the key fails to operate as normal. (Thus the user has confirmation that the key is pressed hard enough to operate, that it hasn't bounced, and that it has had, or has not had, the intended effect.)
- Pressing a normal key appends it to the display.
- In a security application such as entering a password or PIN, pressing any keys (say, [4] [7] [9] [1]) would display an unrelated mask such as `****` so other people cannot read the password the user is keying. In a high security application, there may not even be per-key feedback that indicates how many keys have been pressed.
- Pressing [DEL] deletes the previous normal key.
- If a key press has been processed, nothing retrospectively changes the meaning of the key press. For example, if the display is full (say showing `1234`) then pressing a further key (say [5]) does not change the display (for instance, to `2345`, losing the first key press).

19 ► Define exceptions to its laws. For example, if the display is full, extra keys may be ignored. Perhaps pressing [DEL] deletes the last key in the display, not necessarily the last key pressed.

20 ► *The user need not be aware of specific exceptions to the laws*. Hence, a dependable UI ensures exceptions, including timeouts, are *always* very clear to the user, for example, through color changes (for instance, by dimming invalid key presses; we know this also speeds up users [15] since it shows users what their options are), physical/tactile key feedback, and beeping. Conversely, it must provide feedback on valid key presses (that they have been hit, that they haven't bounced, etc).

21 ▶ In particular, indicate when [OK] would enter invalid data for the application. For example, the user has pressed [CLEAR] [•]—a digit is required before this can be submitted.

22 ▶ If a key press would cause divergence (in the current state), it does nothing. In particular, [OK] can only pass well-formed data to the application.

Devices have different sorts of keyboards:

23 ▶ On a PC or other device with a general-purpose keyboard, very little can be done to provide key-specific feedback before a key is pressed. Keys should beep distinctively if pressing them causes divergence, and otherwise click (to confirm that they have been pressed, and if so, how many times).

24 ▶ On special purpose hardware or on semi-soft keypads (where there are physical keys adjacent to dynamic key legends), keys can be lit if they are enabled, and dimmed and even physically locked if pressing them would cause divergence.

25 ▶ On soft keypads (e.g., on touch screens), diverging keys need not even be displayed.

We want a UI where a user paying sufficiently close attention to it will always be able to obtain their intended effects. We want a UI where pressing [CLEAR] ... *any non-diverging sequence* ... [OK] enters what the user predicts without confirming with the display. Since this law of behavior may be learnt by the user even if not explicitly stated, it is imperative that divergence is indicated audibly and using tactile feedback through the keys themselves, in addition to any visual indication (to which the user may not be paying attention). We thus aim for a UI that is either predictable or beeps (where "beeps" means to make attention-getting feedback, typically a sound) *and* if it beeps the user action causing the beep is otherwise completely ignored.

Divergence can be detected by the system, for instance when a user keys more than can be displayed. A dependable system must alert the user (perhaps by sound) of a divergence or to enable timely and appropriate action. Of course, in some situations, alerts may counter-productively escalate the error rate by stressing the user, or perhaps by causing bystanders to intervene and cause different sorts of problem.

4 A new approach to keyed data entry

We suggest the term *C×CUI*, a correct by construction UI, a UI that is consistently designed and implemented as an instance of a class of related interfaces sharing common properties. We prefer the term C×CUI to UIMS (user interface management system), as a UIMS typically prioritises flexibility and generality, whereas a C×CUI prioritises dependability and rigor: the approach is flexible, but it is not in any way arbitrarily programmable, as UIMS commonly are. Carefully considering and automatically checking the general and abstract properties of UIs leads to better UIs. Consistency, easier iterative design, improved dependability and so on are consequences of the approach.

C×CUIs of a class have specific interaction properties and hence considerably reduce transfer errors and improve user learning times between devices and systems in the class. One would envisage all data entry on a particular device to share the same properties. Additionally, C×CUIs

are easier to modify *while maintaining dependability and clear definition*; they support iterative design well.

Our particular approach is based on finite state machines (FSM) using regular expressions (regex) as a notation to specify them. This approach is not novel *per se* in that the implementation concepts are well-described in 1980s textbooks [1] and, indeed, much earlier in the research literature. The idea may be compared to a "dependable *interactive* Lex" —Lex being the 1975-vintage non-interactive lexical analyzer-generator [12]. The regex is used to specify the syntax of the data input language, and the C×CUI compiles it into a finite state machine (FSM) that is interpreted by a UI virtual machine as a push down automaton (PDA). We note that FSMs and PDAs are simple, well understood and easy to analyse [15]. Following Peter Ladkin, we assert that **if it is possible to use a FSM, then there are overwhelming reasons to use one.**

FSMs are as fast as one can make any system, they can be compiled to simple, provably-correct, hardware to build UI controllers, and they can be shown to have no run-time errors. Of course, FSMs of non-trivial systems tend to become large and for practical use in interaction—where a user is expected to model the UI—we do not just want an FSM, but a *simple* FSM, for some meaning of simple. We assert that a user will find simpler FSMs simpler to understand.

Since keyed data entry retains a history of the user's keystrokes, a strict FSM would need an exponential number of states on the length of the history; hence following [18], we need to factor the UI into simple FSM-like behavior and a separate history. For our purposes of dependable keyed data entry (as we shall show), it is sufficient to for the memory to be the display, and moreover the display will behave like a stack and the state of the FSM will be determined only by the contents of the display. In the case that the display overflows, we consider the display to be a window onto the stack in a way that is determined by design requirements.

There are many ways to represent FSMs; we use the JavaScript object notation, JSON (see www.json.org; [15]) as this is very portable, and it naturally allows the FSM to be combined with other parameters we need that the UI VM implements.

4.1 The regex notation

Programming languages usually distinguish between the values they operate on and their notation; thus literals usually have a clear syntax within the language. Unfortunately when they are string processing languages there is a temptation to elevate the notion of literal so that the operational notation is secondary and becomes a notation within strings. Regular expressions are the classic example: because regex match strings, it seems natural for the regular expression `abc` to match the string abc; but then operators (say, Kleene star) are ambiguous: does `abc*` mean match abc followed by star, or ab followed by any number of stars? Designers of regex of tend to make *ad hoc* decisions: thus `*` might be Kleene star, but `(` might need escaping if is a bracket rather than the literal symbol. Since regex are widely used in user interfaces, such design decisions have been driven by conciseness—which comes at the expense of losing redundancy, clarity and ambiguity. (Macro processing languages, TEX, troff, *Mathematica* represent other examples, to say nothing of the multiple regex sublanguages of Perl, PHP *et al.*)

In most regex notations, there are two sorts of character: literal symbols and operators ("meta-characters"). This causes many readability problems, for example if you want to recognise a symbol that happens to be an operator then the operator has to be escaped. Comments in conven-

tional 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 regex 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.

Since developers want control depending on the context of use, many regex notations have parameterization achieved by using a list of partly-mnemonic letters like `i` to ignore upper/lower case distinctions, often in the form, */pattern/modifiers*, and often the regex may be parameterized programmatically as well, so its properties may even be dynamic and distributed around the program. By "programmatically" we mean that features are implemented in the structure of the coding, use of conditionals and so forth, and therefore (unless it is very carefully documented in comments), the behavior of the program is implicit, typically implemented differently in different parts of the UI [16], and not obvious from the static meaning of the code. In our approach, the behavior is parameterized with features (e.g., whether the display should be masked for password entry; whether there is an delete character key, etc), but the features are explicit. Features are *only* introduced statically in the specification by a list of fully explicit names and values.

For example, `value-display-size` if a feature that sets the size of the display. When the user's input exceeds the display length, the UI goes red and the display shows (unless otherwise specified) `>>>>>>`; the UI then counts keystrokes to ensure DEL is implemented properly. Thus a user operating the device with "their eyes shut" (i.e., not paying full attention) will find that n key presses followed by DEL is *always* equivalent to the prefix of $n-1$ key presses.

When a specification is compiled, a summary of features and feature interaction, if any, is provided. For example autocompletion may 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). Or, autocompletion and masking is unusual—masking is usually used so a bystander cannot see a password, and autocompletion will automatically enter the password with no typing needed! In the case of errors (e.g., that no input is ever required from the user, perhaps because of autocompletion), the compiler rejects the specification.

We introduce and motivate our 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 W3C should be in a leading position to use best practice, but their approach is manifestly *ad hoc* and arbitrary. They repeatedly re-implement very similar parsing strategies; they interleave syntax, semantics and error detection in obscure ways. Their notation does not help. They use words, including "fail" and "abort," in undefined ways, *and* the approach fails with input errors!

Of course, we know that a well-structured formal specification is superior to natural language for all sufficiently complex programs. What is surprising is that the W3C did not take advantage of any formal notation. However, regardless of the issues surrounding the W3C's choice of an informal and incorrect specification, the main purpose of this section in this paper is to use this case study as an interesting example to help define our notation, and to show it is adequate for specifying things that sometimes are expressed in more verbose and obscure ways. That the W3C is by no means the only source of bad number parsing specifications is discussed at greater length elsewhere [14, 16, 17]. Translated into our notation, the W3C specification becomes:

```
1: whitespace* (-: skip any whitespace
2: [ minus ]
```

```
 3: digit+
 4: [ dot digit+ ]
 5: [ exponent-symbol
 6:   [ (-: W3C number can end after exponent!
 7:     [ plus | minus ]
 8:     digit+
 9:   ]
10: ]
11: garbage-characters* (-: ignore anything
```

This is notably shorter and easier to read, and 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 (see discussion below), 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).

Line 1 uses `whitespace`, a variable defined elsewhere, to represent blank text. The name `whitespace` is followed by the postfix operator `*`, which means "match zero or more occurrences" (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 brief line corresponds to W3C's prolix, "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." 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`.

Definitions such as `exponent-symbol = "e"|"E"` can be placed anywhere. Naturally, the system reports errors such as recursion, as well as redefining names or defining names that are not used (or transitively are only used in definitions that are not used).

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

So, what have we achieved?

26 ▶ Brevity and clarity—bringing their usual advantages.

27 ▶ 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.

28 ▶ 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.

29 ▶ 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.

30 ▶ The language compiles into a simple finite state PDA, which can run the UI specified. The W3C has to be converted (unreliably!) by hand.

In short, our regex notation has the power and generality that the W3C felt the need to fall back on the flexibility of English to achieve but it is concise and rigorous. After criticising the W3C floating point definition because of its inscrutability and weak error management, we now propose a better specification:

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

Of course we still require definitions for the lexical terms, `blank`, `sign`, etc. (not shown here), and once a number has been accepted as correctly formed, it has to be converted to the value it represents. The W3C convert characters to a number value *as* the number is parsed, making it even more obscure.

In contrast to W3C, our treatment of signs is and is *obviously* 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 so cannot finish, confusingly, with more numbers—this avoids the W3C mess of permitting data like `2.3E.4`, `1EE8` and `2.3.4`.

4.2 Autocompletion

Autocompletion sounds like a simple feature to reduce keystrokes. However, it introduces design trade-offs. It is not clear whether reducing keystrokes (since some are automatically provided) reduces error rate, or whether the changing mode of the UI (increasing user uncertainty in how much to key) increases error rate. In some cases, keying something that is autocompleted might lead the next keystroke to do something else—thus potentially offsetting the benefit gained by keying less. Autocompletion reduces redundancy: rather than having to get, say, ten keystrokes correct, maybe only two are required; one might then enter the wrong data accidentally more easily, and ironically the "wrong" data would now be well-formed and harder to detect by the application.

Various types of autocompletion feature are provided.

Suppose the regex is `"MONDAY" | "TUESDAY" |...` then the user has to type exactly [M][O][N][D][A][Y], [T][U][E][S][D][A][Y]... to enter data. If **strict autocompletion** is on, then whenever the only option is determined, the user does not need to do it. Thus, the user need only key [M], [T][U], [W], [T][H], [F], [S][A], or [S][U]; the language has been changed to `"M" | "TU" | "W" |...`

With **relaxed autocompletion**, the system also allows the user to key in the full data. Thus [M] is sufficient, but [M][O], [M][O][N] and so on are also allowed; the language has been changed to `"M" ["O" ["N" ["D" ["A" ["Y"]]]]] |...`

Both forms of autocompletion create a UI inconsistency: a unique abbreviation for Monday or Friday has become a single key, but a unique abbreviation for Tuesday or Saturday requires two keys. Hence **minimal autocompletion** sets a lower bound on the minimal abbreviation length. If the length is 3, then the user must key 3 keys: [M][O][N], [T][U][E], etc. This is clearly more consistent. If relaxed autocompletion is also permitted, then the user can key [M][O][N], [M][O][N][D], [M][O][N][D][A] or, in full, [M][O][N][D][A][Y] too, but the sequences [M] and [M][O] are too short. Minimal autocompletion can be set by `value-minautocomplete=3`, for example.

Finally, with **end autocompletion**, the UI only autocompletes when it can complete to the very end of the input. Consider a specification for a credit card number, `four-digits "-" four-digits "-" four-digits "-" four-digits`. Here, as soon as the user keys the first four digits, the system can autocomplete a dash, then waits for the user to key more digits. With end autocompletion, autocompletion does not happen because the autocompletion does not complete the user's input. Thus, a rule for end autocompletion is that if autocompletion occurs, the user can always successfully press OK. End autocompletion also avoids ambiguities in relaxed autocompletion where a key might be either a relaxed continuation or a continuation after the autocompletion.

4.3 Prompting

Many interactive systems provide the user with prompting about what they should key. We consider prompting as a special case of autocompletion: if an autocompletion would be **-**, then this could be a prompt—the user has to key it to proceed (supposing autocompletion is not enabled to do it for them); secondly, if any of several characters could complete the user's input, then show a generic continuation, such as **_**.

This idea supports automatic generation of prompts. Given the regex `digit digit "/" digit digit "/20" digit digit`, the initial prompt would be **__/__/20__**, and as the user keys digits, the display would change to **1_/__/20__**, **12/__/20__**, and so on.

In general we need more flexibility, particularly to permit prompts that are explanatory rather than literal presentations of what the user has to enter anyway. Also, there is no reason to display prompts just inline within the display the user is keying into: other positions, such as above the display or as a tooltip are also possible.

When a regex is prefixed by *p* `>>` it is prompted by *p*, a string or a variable. Thus `"d" >> ("0"|..."9")` prompts digits by **d**. The prompt is the same as the autocompletion that would have been generated had all transitions in the regex been the prompt. If we wanted a **DD/MM/20YY** style prompt, this would be achieved by `("D" >> digit digit) "/" ("M" >> digit digit) "/20" ("Y" >> digit digit)`. With this regex, it would be sensible to have autocompletion switched on so that the slashes can be provided automatically.

Prompts can be applied to regex in various ways and prompts can override each other to create powerful combinations. Thus `"Date?" first >> none >> date` defines a prompt that is **Date?** if the user has keyed nothing, and then changes to nothing (no prompt at all) when the user keys anything. A more complex example is:

```
"Date?" first >> day "/" month "/" year
        day = "Day" only >> digit [digit]
        month = "Month" only >> digit [digit]
        year = "Year" only >> digit digit digit digit
```

Here, the initial display will be **Date?**, but as soon as the user keys a digit, 1, say, the display will change to the more specific **1/Month/Year**, and so on.

Prompting raises interesting issues. First, errors are possible, and they are detected by the C×CUI. For example, `("a" >> "x") | ("b" >> "y")` has inconsistent prompts—the prompt cannot be *both* `a` and `b`. If prompting is combined with autocompletion, this may be confusing, and the C×CUI warns of the potential feature interaction.

4.4 Time-outs and interruptions

Users may be interrupted when entering data. The most common approach to handling interruptions is to time-out the UI in some way, but perhaps subtly in a way that a user may not notice, as done in the Graseby syringe pump that silently times-out without any warning [14].

In our approach, a flashing time-out icon is shown, and its rate of flashing increases (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.

4.5 Error correction

The number regex of section 6, below, does not permit a number to start with a decimal point. If the user keys [CLEAR] [•] [1] [2] then the C×CUI treats this as an error. Instead, the C×CUI could error correct the user's input to `0•12` by inserting the missing zero.

In our view, in a dependable UI, if the user makes a slip, it is better to tell the user that a slip has been made and allow the user to reflect and correct the error, rather than to make a correction that is potentially the wrong one. For example, if the user keys [CLEAR] [•] [1] [2] *a priori* we do not know whether a zero was omitted or that the decimal point is an error, possibly a transposition with a later key—the user might have intended to key [CLEAR] [1] [•] [2], or any number of other things. A dependable UI should not attempt to mindread *especially* after a recognized error, which is a symptom of the user not doing what they intended, and thus "mind reading" is itself intrinsically error-prone!

5 Formal summary

Let $\mathscr{L}$ be the language specified by the regex; for example if the regex is `"a"["b"]"c"` then $\mathscr{L} = \{abc, ac\}$. Even though Kleene star may be used, all strings in the language are no longer than a defined constant, *max*. Define $\wp\mathscr{L}$ to be the set of all prefixes of $\mathscr{L}$; in particular $\mathscr{L} \subseteq \wp\mathscr{L}$. Hence for $\mathscr{L} = \{abc, ac\}$, $\wp\mathscr{L} = \{\varepsilon, a, ac, ab, abc\}$. The empty string, ε, is a prefix of any string.

If $\mathscr{L}$ is empty (e.g., all strings in the regex are longer than *max*), then this is a compile-time error since the user can do nothing.

The display shows what the user has keyed, σ. Divergence occurs if $|\sigma| > max$. Pressing a key [k], if the key is highlighted, changes the display to σk. Pressing [DEL], if [DEL] is highlighted, changes the display σK to σ. Pressing [CLEAR], if [CLEAR] is highlighted, changes the display to ▀, $\sigma = \varepsilon$ (i.e., nothing)

The display is colored as follows:

Amber	if $\sigma \in \wp\mathscr{L} \wedge \sigma \notin \mathscr{L}$ (more input is required)
Blue	if $\sigma \in \mathscr{L}$ (input is syntactically well-formed but may or may not be accepted by the application)
Green	if $\sigma \in \mathscr{L}$ and the input has been accepted by the application
Red	if $\sigma \notin \wp\mathscr{L}$ (input is not a prefix of any possible string), or if the input string length exceeds the display size

Coloring we use derives from [16], independently proposed in 2001 [6]; by being more aware of keying errors users should be able to enter data more reliably. Thus if the user's input is syntactically incorrect, the color is red; if it is correct, it is blue; if it is a prefix of a correct form, it is amber. Finally, if the user's input is equal to a value previously accepted by the application, the color is green. Colors are also augmented with sound and icons. A slightly simpler approach is to combine the green and blue colors, but it is usually useful to distinguish between synactically correct input (blue) with actually accepted input (green), and using green alone would potentially give the user the incorrect impression that syntactically correct input was semantically correct—when the display goes green, the user might be tempted to press OK. In the chosen design, blue does not tempt the user, and the display goes green *after* the user has pressed OK with syntactically correct data. As usual, colors are not pure, to accommodate for common color blindness issues.

Optionally, the UI can be re-colored by the application on each keystroke—for example, the regex specifies well-formed numbers, but the application additionally requires the numbers to lie within a certain range. (Such a restriction can be specified in a regex but it is tedious, and does not allow the dynamic behavior of responding to the application's context.) However, the application is only allowed to make the color "redder" as it is not allowed to correct errors the user has made. In the present paper, we only consider "incorrect" to mean syntactically incorrect, but the application can be passed blue data every time it changes and possibly colors the UI red if the data is out of range or invalid for any other reason.

The following keys are highlighted (enabled): $\{c|\sigma c \in \mathcal{PL}\}$; equivalently, these keys are in the first set of the language. The DEL and CLEAR keys are highlighted iff $\sigma \neq \varepsilon$. The OK key is highlighted if the display is green.

The UI beeps when the display changes to red or a non-highlighted key is pressed.

6 Example: Dependable keyed number entry

The Institute of Safe Medication Practices informally defines safer ways of writing numbers [16] to avoid potential confusions such as `5` and `.5`, etc, 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 [16] for more details. The specification for the ISMP rules is as follows:

```
name: "ISMP numbers"
features: no-display-prompt display-left-align
          value-buffer-length = 6 (-! 6 for demo
input:    (zero-digit | nonzero-digit any-digit*) [dot fraction-part]

any-digit = zero-digit | nonzero-digit
fraction-part = any-digit* nonzero-digit
zero-digit = "0"  dot = "."
nonzero-digit = "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

The `name` string is required to allow the C×CUI compiler to generate several UIs. It is helpful to generate visualizations of regex (e.g., figure 1); such visualization helps developers see their

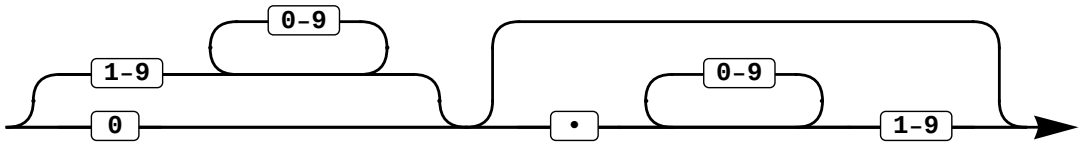

Figure 1: A "railroad" syntax diagram automatically generated from and visualizing ISMP numbers. For many tasks, railroad diagrams may be more readable and more suitable for users than transition diagrams.

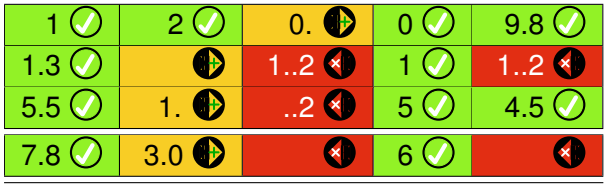

Figure 2: How traffic lights and cell coloring (possibly not very clear in monochrome reproduction in this paper!) might be used in a spreadsheet, here for totalling columns of three numbers. In case of color blindness, the green light is shown distinctively, and, in fact, the "green" is a teal (green-blue) to help common green-confusing color deficits. Coloring and traffic light choice here is as defined by ISMP rules.

specifications more clearly, and hence helps detect typos or other mistakes or misunderstandings in them [5].

The ISMP rules could be used in spreadsheets, and this illustrates the extra information traffic lights provide when there are many data entry fields and underlying application modes. A spreadsheet is the most familiar example of multiple data entry fields, some of which display data entered by the user and some show computed results computed. Figure 2 shows how spreadsheets could use traffic lights to show the validity of both user input and calculated output (here, the totals of columns of numbers). A column total is green if all its parts are green, red if any of its parts are red, and amber if any is amber and none are red. In general, of course, a spreadsheet could perform any calculation: a formula's cell is formatted red if any cell it refers to is red, etc.

7 Future work

We are currently working on a new literate programming implementation [11, 13], which will improve the quality and opportunities for wider dissemination.

A regex specifies what is accepted, so it cannot distinguish between different sorts of error, since errors are not explicitly specified. If we labeled regex with (say) `error` we could use prompts to discriminate between different sorts of error. For example: `error: "Can't have more than one decimal point" >> dot [!dot] dot.`

Note that derivatives [7] allow the easy implementation of regex with boolean operators, in-

cluding complementation.

Reading keyed data is generally only one part of a UI; buffer automata [18] are a generalised form of state machine where states can be buffers, such as those as specified in this paper, and they thus provide a more general approach to implementing dependable UIs while retaining the benefits discussed in this paper.

Fixing bugs in safety-critical or potentially safety-critical domains does not need evaluation! However the use of traffic lights and time-outs, while apparently sensible, needs empirical evaluation before deployment in safety critical environments; what can be evaluated is the use of the traffic light indicators to raise perceptual awareness, to see whether it reduces undetected errors. Since features are variables, they may be used for experimental controls (e.g., [3] explores lock out), not just for evaluation; also, it is trivial to generate sets of UIs differing in details for comparative evaluation of features. Parameters are also provided for injecting errors, which can simulate key bounce, transpositions and so forth.

8 Conclusions

UI design involves trade-offs, but this paper has shown definitively that even a simple 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 and invariants 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—sadly, almost all the time—UIs will (eventually) induce adverse incidents as a result of heedless mismanagement of error.

The approach presented in this paper generates 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 notice and manage errors. The C×CUI presented in this paper together provide an unusually convenient and consistent UI for dependable keyed data entry.

A prototype implementation is at harold.thimbleby.net/regex. The examples in this paper work on HTML5 compliant browsers, including the iPad, where it looks and feels like a device.

Acknowledgements

Ann Blandford, George Buchanan, Paul Cairns, Abigail Cauchi, Andy Gimblett, Robin Green, Michael Harrison, Patrick Oladimiji, Brian Randell. Funded by EPSRC Grants EP/F020031/1 and EP/G059063/1.

Bibliography

[1] A. V. Aho, R. Sethi & J. D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] A. Avižienis, J-C. Laprie, B. Randell, C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable and Secure Computing*, **1**(1):11–33, 2004.

[3] J. Back, D. P. Brumby, A. L. Cox, "Locked-out: Investigating the effectiveness of system lockouts to reduce errors in routine tasks," ACM CHI, 3775–3780, 2010.

[4] N. Bevan, "Extending quality in use to provide a framework for usability measurement," Lecture Notes in Computer Science, **5619**:13–22, 2009.

[5] A. F. Blackwell, "SWYN: A visual representation for regular expressions," in H. Lieberman, ed., *Your wish is my command: Giving users the power to instruct their software*, pp245–270, Morgan Kauffman, 2001.

[6] C. Brabrand, A. Møller, M. Ricky, M. I. Schwartzbach, "PowerForms: Declarative client-side form field validation," *World Wide Web Journal*, **3**(4):205–214, 2001.

[7] J. A. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM*, **11**(4):481–494, 1964.

[8] R. Feynman, "Cargo Cult Science," *Engineering and Science*, **37**(7):10–13, 1974.

[9] ISO/IEC Standard 14977:1996(E), *Information technology —Syntactic metalanguage—Extended BNF*, www.iso.org, 1996.

[10] C. Jones, P. O'Hearn, J. Woodcock, "Verified software: A grand challenge," *IEEE Computer*, 93–95, 2006.

[11] D. E. Knuth, *Literate Programming*, Stanford University Center for the Study of Language and Information, 1992.

[12] M. E. Lesk, E. Schmidt, *Lex—A lexical analyzer generator*, Bell Labs, 1975.

[13] H. Thimbleby, "Explaining code for publication," *Software—Practice & Experience*, **33**(10):975–1001, 2003.

[14] H. Thimbleby, "Interaction Walkthrough: Evaluation of safety critical interactive systems," Lecture Notes in Computer Science, **4323**:52–66, 2007.

[15] H. Thimbleby, *Press On*, MIT Press, 2007.

[16] H. Thimbleby, P. Cairns, "Reducing number entry errors: Solving a widespread, serious problem," *Journal Royal Society Interface*, **7**(51):1429–1439, 2010.

[17] H. Thimbleby, "Interactive systems need safety locks," IEEE ITI International Conference on Information Technology Interfaces, pp29–36, 2010.

[18] H. Thimbleby, A. Gimblett, A. Cauchi, "Buffer Automata: a UI architecture prioritising HCI concerns for interactive devices," ACM Engineering Interactive Computer Systems, 2011, in press.

[19] Tripwire Team, *Truly useful form validation scripts for front end development*, www.tripwiremagazine.com/2009/11/truly-useful-form-validation-scripts-for-front-end-development.html, 2009.

[20] World Wide Web Consortium, A vocabulary and associated APIs for HTML and XHTML W3C Working Draft, http://www.w3.org/TR/ html5/common-microsyntaxes.html#real-numbers, 2010.