Buffer Automata: A UI Architecture Prioritising HCI Concerns for Interactive Devices

Harold Thimbleby
FIT Lab
Swansea University
h.thimbleby@swansea.ac.uk

Andy Gimblett
FIT Lab
Swansea University
a.m.gimblett@swansea.ac.uk

Abigail Cauchi
FIT Lab
Swansea University
csabi@swansea.ac.uk

ABSTRACT

We introduce an architectural software formalism, *buffer automata*, for the specification, implementation and analysis of a particular class of discrete interactive systems and devices. The approach defines a layer between the physical user interface and the application (if any) and provides a clear framework for highlighting a number of interaction design issues, in particular around modes and undo.

Author Keywords

Buffer automata; modes; undo; interaction programming; structural usability.

ACM Classification Keywords

H.5.2 (D.2.2, H.1.2, I.36) User Interfaces: Theory and methods

General Terms

Design, Theory

1. INTRODUCTION

General purpose programming languages can implement any interactive system, but such systems are not easy to analyse for their interaction properties. Conversely, finite state machines can in principle also describe any interactive system—typically as *enormous* FSMs—but for non-trivial systems they are impractically large and have no clear structure to support insights into human computer interaction issues. In this paper we introduce an extension to FSMs which aims to address this state explosion issue, at least for some systems.

Consider something as simple as a handheld calculator: it has a few basic modes (off, adding, subtracting...) but tracking its states means also tracking (say) 10^8 possible numbers for its display, 10^8 for its memory, 10^8 for the working number, and perhaps 10^8 for any constant (when pressing $\boxed{=}$ just adds the current number to the constant). Its full state space thus has at least 10^{32} states; this simple device is at the limits of what can conveniently be handled by current model checking technology (e.g., by exploiting its symmetries), but human users obviously do not conceptualise a state space of this size. Rather (we suggest) users do think about interactive systems a bit like FSMs, but only in terms of their *mode space*, and the exact number values handled by the calculator are abstracted away in the user's mental model—indeed, such abstraction is much of the purpose of the device.

We propose that many simple (but non-trivial) interactive systems can be thought of like this, that is as a manageable space of a few explicit modes combined with a collection of abstracted data values, and that this captures the structure users are aware of and can manage, and abstracts away the structure users ignore details of. There remain many complex interactive systems beyond the reach of such an approach, at least as proposed here (but see section 6); we are particularly concerned at this time with the many devices, from calculators to medical infusion pumps, that we *can* handle clearly and rigorously. Such 'simple' systems still have non-trivial interaction issues, and the approach proposed here exposes and makes explicit some of these issues—and even where it does not resolve them, it provides a framework to support clearer arguments about design choices.

This paper makes the following contributions: we introduce and define *buffer automata*, a new formalism for structuring and thinking about user interfaces; we provide abstract and concrete examples of buffer automata in use; we show that buffer automata provide a new perspective on several issues of interaction design and programming, including modes and undo, and in particular we describe how to automatically compute modes within the formalism. It is our hope that this paper will direct and stimulate further discussion and exploration of some of the issues raised.

We present a review of potentially alternative and contrasting approaches in Section 5; for a short paper, the background literature is perhaps best presented after we first emphasise and examine the focus of our contribution.

2. BUFFER AUTOMATA

As discussed in Section 1, many interactive systems may be thought of as a relatively simple abstract mode space together with various data values which the user may alter. Thinking of the mode space as a 'classic' FSM or automaton, and the data values as being held in buffers where they are manipulated then leads to the idea of a *buffer automaton*.

Buffers are data values the system keeps track of, such as phone number, radio station, CD track, time now, alarm time, drug infusion rate. *Modes* are significant states of the system, such as on, off, playing, infusing, not infusing, standby. Among other things, modes dictate which buffers are receiving user input, and which are exposed as output. Thus, user actions give rise to input events which are distributed to buffers according to the current mode, modifying buffer values and potentially triggering mode changes. (This mechanism, and the exact relationship between buffers and modes, is explored in Section 2.1.)

In principle a user is generally aware of the current mode(s) of the system and the available actions to change modes; conversely, users do not track exact buffer values, but may know in principle whether they are 'right,' 'wrong,' 'nearly right,' etc. Users rely on systems to keep track of buffer details unless actually interacting with a particular buffer. For example, consider a nurse on a hospital ward with patients connected to drug infusion pumps, asked "is this patient getting an infusion?"; the nurse knows the answer is yes, because earlier they put the device into the infusing mode; when asked "how fast is the patient being infused?" the nurse will probably check the device. Here 'infusing' is a mode, but 'infusion rate' is a buffer.

Another example: a radio can be on or off, and can be tuned in to FM or MW bands; it has 100 possible volume levels and 100 possible tuning frequencies; it responds to events $\boxed{\text{ON}}$, $\boxed{\text{OFF}}$, $\boxed{\text{MW}}$, $\boxed{\text{FM}}$, $\boxed{\text{VOLUP}}$, $\boxed{\text{VOLDN}}$, $\boxed{\text{TUNUP}}$, $\boxed{\text{TUNDN}}$. Modelling this as a 40,000 state FSM ($2 \times 2 \times 100 \times 100$) certainly obscures its structure; modelled as a buffer automaton we have a space of four modes (off/MW, off/FM, on/MW, on/FM) and some independent buffers with simple, well understood behaviour (e.g., for the volume level). Events are cleanly distributed to the various buffers according to mode (when off, everything but $\boxed{\text{OFF}}$ is ignored, say), and there is no interaction between the buffers in this highly orthogonal device. Variants are possible and explored later in the paper.

Tasks on many systems require a pair of activities: set up one or more buffers (such as the rate of infusion), and make the system do the action with a buffer value (such as perform an infusion). Such systems are well suited to modelling as buffer automata. Of course, any sufficiently complex component could model or even implement an entire system—nothing *needs* to be partitioned into independent modes and buffers. Our proposition is that buffer automata can provide clear descriptions of interactive systems of a particular class, amenable to particular kinds of analysis: the utility of the formalism is its exposure of certain HCI issues such as modes and undo, and as an attempt to model systems closely in terms of how users think of them.

2.1 Buffers and modes

As described above, a buffer automaton is an FSM-like mode space accompanied with a set of buffers (whose structure we define shortly). In fact, while it is possible and may be useful to model the mode space explicitly, we see modes rather as an emergent implicit feature, arising from how inputs are

distributed to buffers—thus, the mode space is to be computed, not specified directly. Let us explore this subtlety.

Consider the radio example above. There are on/off and MW/FM 'modes,' combining orthogonally to form a mode space with 4 states; it is precisely the states of *this* space that we refer to when we say *mode*. On/off and MW/FM are not themselves modes: like volume and tuning, they are buffers—data values the user can manipulate—but they are special buffers (only) in that they introduce mode behaviour: they influence how input events are interpreted. This interpretation of the word 'mode' is well-founded: according to [6], a mode is a "*context that changes the interpretation of commands*"; similarly, [11] says "*for any given gesture, the interface is in a particular mode if the interpretation of that gesture is constant*". Our notion of *focus* formally captures this concept—see Section 2.3.

So on/off and MW/FM are just buffers; they happen to have very simple structure, and we can easily model them as 2-state FSMs, but in general this need not be the case: buffers can have rich structure, and even rich buffers may influence mode. Suppose the radio also has a $\boxed{\text{MUTE}}$ button (another 2-state buffer), which is ignored if the volume is 0—then the volume buffer (modelled as a 100-state FSM, perhaps, corresponding to a physical slider or knob) also contributes to the radio's mode. Every buffer we've seen so far is easy to model as an FSM, but this need not be so (a text entry box is a good counterexample: it *can* be modelled as an FSM, but not naturally), so our definition of buffer, below, is necessarily quite general; 'rich' buffers such as text entry boxes can still influence the system's mode, but perhaps with a greater risk of confusion for the user.

2.2 Definition: Buffers

A buffer is essentially an object for some data along with several functions for manipulating and accessing that object. In order to support undo (see Section 2.5), we structure the object as a *history* rather than a single value, formed from an initial value and a sequence of modifications (perhaps, but not necessarily, corresponding to input events); it is easy to ignore this history if desired, as we shall show by example.

Let *FinSeq*(X) be the set of finite sequences of elements of X. Given two sets X and Y, the set of *histories* over X and Y, $\mathbb{H}_{X,Y}$ is the set $X \times FinSeq(Y)$. That is, a history has an initial value in X, followed by a sequence of values in Y. Notionally, the Y values modify the initial X value in some sense, although various interpretations are possible (including where $X = Y$ and 'modify' means 'replace').

A buffer is a tuple $(H, \delta, \lambda, \phi)$, with:

- $H : \mathbb{H}_{C,I}$ is the buffer's *history*, where C is the buffer's *contents alphabet*, and I is the buffer's *input alphabet*.

- $\delta : \mathbb{H}_{C,I} \times I \to \mathbb{H}_{C,I}$ is the buffer's *input function*, which modifies the buffer's history in response to the buffer receiving input events.

  We argue that there are a small number of 'off the shelf' components which can be combined to form all reason-

able input functions. For example: *append* to the history; *delete* the history's last element; *reset* by clearing the history; *forget* by replacing the initial value with the result of λ and then performing a *reset*. Additionally, functions like *number*, *date*, *name*, *currencyValue*, etc, provide consistent ways of handling application values.

- $\lambda\colon \mathbb{H}_{C,I} \to C$ is the buffer's *access function*, mapping the buffer's history to some value in C, such as a floating point number. This might perform a fold over the history, or perhaps access its last value, etc; again, a small number of standard functionalities could conceivably cover most real buffers.

- $\phi\colon C \to \mathbb{H}_{C,I}$ is the buffer's *update function*, through which the application in which the buffer is embedded may modify its history and thus its value (see Section 2.6).

FSMs can easily be modelled using this mechanism, but many other structures are possible.

Example: Numerical buffer with direct manipulation.
Our radio's volume and tuning buffers might be represented as follows; let $C = \{n \in \mathbb{N} \mid 0 \le n \le 99\}$, $I = C$. Then: $\forall x, z \in C \bullet \forall y \in FinSeq(C)$:

$$\delta((x,y),z) \quad = \quad (z,y)$$

That is, the initial value is simply overwritten with new inputs, and the history is never used: whatever number is selected is entered directly, overwriting any previous value.

Example: String buffer with clear and delete. Let $C = FinSeq(\{a,b\})$, the set of finite sequences of a's and b's. $I = \{a, b, clear, delete\}$. Let $\delta\colon \mathbb{H}_{C,I} \times I \to \mathbb{H}_{C,I}$ be: $\forall x, z \in C \bullet \forall y \in FinSeq(\{a,b\})$

$$\begin{aligned}
\delta((x,y),a) &= (x, y \frown a) \\
\delta((x,y),b) &= (x, y \frown b) \\
\delta((x,y),clear) &= (x, \langle\rangle) \\
\delta((x,\langle\rangle),delete) &= (x, \langle\rangle) \\
\delta((x, y \frown z),delete) &= (x, y)
\end{aligned}$$

(Where $\frown$ is concatenation.) That is, a and b are appended to the history sequence, whereas $clear$ and $delete$ modify that sequence. Thus, the buffer's history consists of some initial value in C (the empty string, say), followed by a finite sequence of further values in C. Then, let $\lambda\colon C \times I \to C$ be the concatenation of the buffer's history sequence onto its history initial value. This version of the buffer has no maximum length.

2.3 Definitions: Buffer automata, focus and modes
A buffer automaton is a tuple $(\mathcal{S}, L, \mathcal{L}, s_0, \Sigma, \mathcal{F}, \Delta, \mathcal{V})$, with:

- $\mathcal{S}$ is a tuple of buffers. Labels $\in L$ serve as unique names of buffers, and the naming $\mathcal{L}$ is a bijection from labels to buffers. The state of the BA is the state of its buffers; s_0 is the BA's initial state.

- Σ is the BA's input alphabet, the union of each of its buffers input alphabets (which need not be disjoint).

- $\mathcal{F}\colon \mathcal{S} \to \Sigma \to \mathbb{P}(L)$ is the BA's *focus function*, which, given the current state of the BA's buffers, tells us for each input the names of buffers that should receive that input. From $\mathcal{F}$ we derive our formal notion of mode—see below.

- Δ is the BA's derived *input function*, $\Delta\colon \mathcal{S} \to \Sigma \to \mathcal{S}$, which distributes inputs to the BA's various buffers in accordance with $\mathcal{F}$. The definition is omitted here, but is simple: an input is distributed to every buffer which is currently in focus for that input.

- $\mathcal{V}$ is the BA's *visibility function*, $\mathcal{V}\colon \mathcal{S} \to \mathbb{P}(L)$, and determines the names of the visible buffers (and thus what is visible to the user may be computed from their values).

As discussed in Section 2.1, we see mode as a context determining the interpretation of user inputs; assuming (reasonably, for any sane device) that a given buffer's interpretation of its input is always consistent, mode is thus defined solely by the distribution of inputs to buffers—which we can compute trivially given the focus function $\mathcal{F}$. The current mode of a BA is essentially a lookup table telling us, for each buffer (identified by name), which inputs it can receive right now. Thus, we have $mode\colon \mathcal{S} \to \mathcal{L} \to \mathbb{P}(\Sigma)$.

2.3.1 Radio example
Let our radio have the following four buffers:

- *OnOff*, an FSM with states $\{on, off\}$, actions $\{On, Off\}$.

- *Band*, an FSM with states $\{mw, fm\}$, actions $\{MW, FM\}$.

- *Volume*, with $C = \{0, \ldots, 99\}$, $I = \{v_\uparrow, v_\downarrow\}$, volume values modified with up/down actions, under some sensible interpretation of the buffer's history (e.g. perhaps with wraparound, depending on the physical mechanism used).

- *Tuning*, with $C = \{0, \ldots, 99\}$, $I = \{t_\uparrow, t_\downarrow\}$ (similarly).

Sensible initial BA values might be $(off, mw, 0, 50)$, say. Σ is $\{On, Off, MW, FM, v_\uparrow, v_\downarrow, t_\uparrow, t_\downarrow\}$. $\mathcal{F}$ is simple: all events go their respective buffers, unless *OnOff*'s value is off, when *On* goes to *OnOff* and all other events are ignored.

Unlike our earlier conception of the radio, this device actually has only 2 modes: on and off. *Band* has no influence on interpretation of inputs: in effect, we have modelled an old style tuner with a single tuning control for both bands. Instead, let's replace *Tuning* with separate tuning values for MW/FM, better reflecting the operation of a modern radio:

$$\begin{aligned}
MWTune \quad &\text{with} \quad C = \{0, \ldots, 99\}, I = \{t_\uparrow, t_\downarrow\} \\
FMTune \quad &\text{with} \quad C = \{0, \ldots, 99\}, I = \{t_\uparrow, t_\downarrow\}
\end{aligned}$$

Note that *MWTune* and *FMTune* have identical input alphabets; we modify $\mathcal{F}$ so $\{t_\uparrow, t_\downarrow\}$ events are delivered to the appropriate buffer according to the value of *Band*, which is thus now mode-relevant. We still don't have 4 modes however, only 3: off, on/MW, and on/FM. Perhaps that's fine— or perhaps it's possible to turn the volume and tuning knobs while the device is off; a variety of approaches are possible. Even with this simple example we see that thinking concretely about focus and modes brings to light some interesting design questions we might otherwise have ignored.

2.4 Mode space discovery

A buffer automaton's mode space may be discovered using *model discovery* [7] to dynamically explore the state space of all mode-relevant buffers, producing a graph whose nodes are (BA state, focus) pairs; the mode space is then obtained by collapsing together states with identical focus, producing a graph whose nodes are distinct focus values (modes) and whose edges are events which change the mode. Note that the set of mode-relevant buffers must be specified explicitly, otherwise a full exploration of the BA's complete state space would be required just to learn which buffers *are* mode-relevant.

Alternatively, given a suitable representation of $\mathcal{F}$, the mode space can in principle be computed statically. However, as the application may in general modify any buffer freely (see Section 2.6), the statically computed mode space for a given system may be inaccurate, and dynamic discovery is probably safer; at the least, comparing static and dynamic mode spaces tells us something about the BA and application's interaction with each other.

2.5 Undo and history

Common wisdom is that user interfaces should support undo [12]; but, actually, it is more important that they support the user achieving their goals in the face of error. Undo is just one way of recovering from (recognised) error, but often it is not the best strategy. In particular, we argue, undo is often not the right strategy—or not even meaningful—when dealing with mode-changing events. If a nurse accidentally starts an infusion before they intend to, and they notice the error, they wouldn't expect to hit UNDO to fix the problem: they'd hit STOP, triggering another mode change—and that would just be the first step in fixing the error. UNDO probably *couldn't* restore the device to its previous state, due to the physical effect of starting the infusion. Conversely, if the nurse makes a keying error while entering the desired infusion rate, UNDO is a perfectly reasonable approach. This points to another distinction between mode-relevant and non-mode-relevant buffers: the latter can in general use their histories to support undo, whereas the former need not (and often should not); in some cases a mode change represents an action which *can't* easily be undone: that is useful information for designers. This is a rather sweeping generalisation of course, and there will be exceptions; for example, in many simple buffers (such as in our radio), all events have inverses so an explicit UNDO shared between buffers is pointless.

2.6 Buffer automata in context

Conceptually, we see buffer automata as an interpretive and structure-imparting layer between the UI/presentation of an interactive device and its underlying application; after each buffer in focus performs its transition, control is passed to the application to perform access then update operations. The BA is then ready for its next input event. How exactly this is structured is left generic in order to accommodate a wide range of approaches. A typical setting is illustrated in Figure 1. User input is received from the UI and passed to the BA as members of Σ, where it is distributed to various buffers' input functions δ as dictated by $\mathcal{F}$; the application
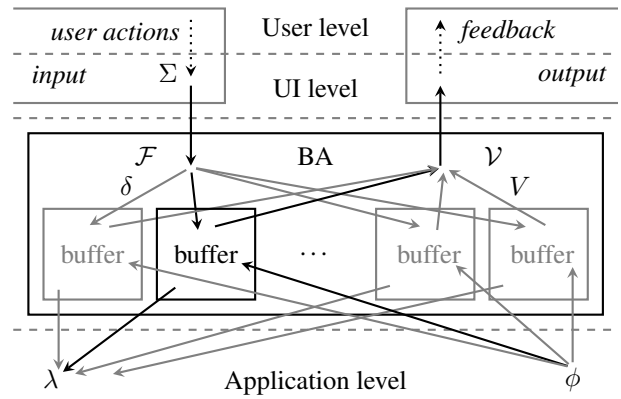
Figure 1. Buffer automata in context. User actions translate to BA inputs in Σ, distributed by $\mathcal{F}$ to each buffer's δ input function; the application reads buffer values via λs and updates via ϕs if necessary; values for feedback are projected by V, filtered according to $\mathcal{V}$.

then has the opportunity to read and update buffer values via each buffer's λ and ϕ functions (as well as dealing with other events, such as non-UI I/O, alarms, etc); buffer values are then exposed, subject to $\mathcal{V}$, as views in the UI using some function $V : \mathbb{H}_{C,I} \to \mathbb{V}$ from the buffer's state to the view domain $\mathbb{V}$. How to handle the display of buffer contents, and the role of the function $\mathcal{V}$—the converse of $\mathcal{F}$—and its relationship with modes remains future work to be explored; we note that 'visibility' might in fact involve non-graphical elements such as sound or vibration.

3. EXAMPLE: PULSE MUSIC SYNTHESIZER

The *Waldorf Pulse* is a 1990s-era electronic musical instrument, controlled remotely for performance via the MIDI protocol, but which may also be programmed via a front-panel interface consisting of 6 knobs, 4 buttons, a 3-character display and 7 LEDs (see Figure 2). There are 69 parameters that control the sound being produced by the instrument (e.g., 'Volume,' 'Arpeggiator Tempo'), where such a collection of parameters is called a *patch*; the Pulse has a memory of 99 patch slots. Patch parameters are edited via the matrix arrangement on the right of the panel: the 69 parameters are arranged in a 6×6 matrix (with some doubling-up—see below), and each of the 6 knobs is dedicated to one column of this matrix. Only one row of the matrix is active at once, as indicated by a red LED to its left and advanced by pressing the large oval Mode button. Finally, the blue Shift button under Mode toggles between parameters where they are doubled up (the red LED flashes when 'shifted'). For example, if the second LED down is flashing, and we rotate the third knob from the left, we modify the 'OSC2 Keytrack' (whatever that means) parameter.

We have simulated the Pulse's patch editing interface using a buffer automaton in Haskell (connected to a web GUI in JavaScript/HTML5 Canvas). Every patch parameter has an associated buffer, and each buffer responds to the events $\uparrow_n$ and $\downarrow_n$ for some column number $1 \leq n \leq 6$. There are also simple buffers for four values associated with modes—see below. We only modelled a single patch, and have not considered the Pulse's memory and related functionality.

Figure 2. Screenshot from running simulation of Waldorf Pulse control panel. Patch control knobs at bottom right. (Images reproduced from Waldorf Pulse user manual with permission.)

Figure 3. Mode space of Pulse buffer automaton; *row* and *shift* form a regular structure of two concentric hexagons, with off in the centre; *Mod Select* introduces six more modes to the right of the hexagons. Note that this graph is nondeterministic (multiple *row* edges at top-right).

Figure 4. Alaris infusion pump mode space computed from the focus function of our BA model. Mode names are implicit, and have been chosen to reflect each mode's role. Some actions (*Opt1*, etc) correspond to soft buttons; a model using displayed names (e.g., 'VTBI') would have a different, possibly more meaningful, mode structure.

Turning a knob should obviously not modify all 6 (or more) parameters in that knob's column. The Pulse's interface is thus very modeful: how turning a given knob is interpreted depends (at least) on which row of the matrix is active, and (in some cases) whether shift is engaged. The mode-relevant buffers on the Pulse are 'OnOff,' 'Row,' 'Shift' and 'Mod Select' (on row 4, knob 3 controls which 'sub-row' knobs 4, 5, 6 are focussed on)—all of which can be modelled as small FSMs.

We used mode space discovery to compute the Pulse BA's mode space (see Figure 3), which is seen to be quite regular. Our Pulse's focus function is hand-written; we plan to devise a domain specific language (DSL) for describing focus, from which the mode space may be computed directly via abstract interpretation, rather than dynamically by discovery; the dynamic version may however remain useful as a correctness check for comparison against the statically computed version.

4. EXAMPLE: ALARIS INFUSION PUMP

The Cardinal Health Alaris GP volumetric infusion pump [2] is an interactive medical device with 14 buttons, designed to provide patients with controlled delivery of drugs. We modelled the device thoroughly using an interactive *Mathematica* program with a realistic graphical animation that allows user testing to confirm the program is an accurate interaction simulation. The simulation has a transition graph model with 4.8×10^{15} states; such a model is unwieldy and computationally costly to analyse.

We recently recast this model as a buffer automaton, with 10 buffers: 3 are for numbers in the range 0–999; 6 are interrelated mode-relevant switches: on/off, running/not running, infusing/not infusing, VTBI Mode on/off, clear mode on/off, and alarm silent, beeping or muted. This multiplies up to $2^5 \times 3 = 96$, but (for instance) when the GP is off, it cannot be infusing or alarming, so in fact only 22 combinations are possible, which may be further reduced to just 6 modes with *distinct* input behaviour; mode space discovery finds exactly these modes—see Figure 4.

5. RELATED WORK

A large number of formalisms take an automata-based approach to modelling systems, though most are more general than buffer automata and do not emphasise interaction and HCI concerns as we aim to. A key example is Statecharts [8, 9] (see also the closely-related UML state diagrams), an automata-based visual formalism with a variety of features enabling the representation of many complex systems using structured diagrams with nesting, composition, abstraction, guarded broadcast messaging and history—all of which yield a rich semantics. In contrast, buffer automata try to be a simple formalism that bridges certain user and device perspectives as cleanly as possible, by refusing to represent some controversial features. Very similar comments might be made for very many related formalisms (Petri nets, CSP, ATNs...): each provides a well-defined generalisation that makes the formalism appropriate for some domain (pro-

cess control, say) but none specifically address human factors or HCI concerns, as is the intention for buffer automata as introduced here. Modechart [10], for example, is a specification language for real-time systems built upon a real-time logic (specifically, RTL). Like BAs, modecharts expose mode explicitly, and aim to aid reasoning about mode-rich systems; Modechart is quite low-level, however, and strongly concerned with timing issues.

Model checking [4] is a well-established technique for automatically verifying a state space model against a logical specification. We have recently seen how this can be applied to interactive systems [1] but this general approach still suffers from the state explosion problem. The BA approach to separating buffers from modes may be compared with the technique of *data abstraction* in model checking. Compositional model checking [3] tries to overcome state explosion by breaking up the model into smaller components; similarly a working theory of hierarchy for buffer automata (see Section 6) would be useful.

6. CONCLUSION

In this paper we have introduced buffer automata, a formalism for describing interactive systems which explicitly separates data entry interactions from mode-changing interactions; we have described how modes emerge from the interaction of buffers via the notion of focus, and thus how to compute a BA's mode space. The formalism introduces a new perspective on the role of undo, and when it is appropriate or not. Imagined and real-world examples demonstrate that BAs are a viable tool for modelling devices of a certain kind in a straightforward and comprehensible manner.

Designers make decisions about UI behaviour, and a BA makes some important decisions explicit. In a conventional program, the UI's structure doesn't relate directly to the user experience; there may be an event loop and UI code sensible to a programmer, but improving the code doesn't necessarily improve the UI or engage with HCI concerns. In contrast, a BA has explicit structures such as the δ function which are *supposed* to be simple. We would argue that a programmer making some δ code simpler improves the user interface design; conversely, programming a Turing Machine inside a δ (which is possible) defeats the object of clarity for the user. By extension, mode-relevant buffers should probably be small and FSM-like to keep the mode space comprehensible.

Future work on buffer automata must explore a number of open issues. The relationship we describe between undo, modes and buffers is convincing but none of the examples in this paper use undo; they are based on real devices, but we should model some other devices which offer undo and check our hypotheses there. Currently, buffer automata have no hierarchical structure; thus, we can't build complex UI elements from simpler ones (as buffer automata), or, realistically, re-use BAs between contexts. One interesting possible line of enquiry here involves positioning BAs as Control components within PAC [5] triads. Similarly, in our current arrangement buffers cannot modify each others' contents or

state except via the (analytically opaque) application layer; a working theory of hierarchical buffer automata could possibly overcome this—though we also argue that this is often in fact a desirable feature. Timeouts and other external events could warrant similar treatment. As mentioned in Section 3, we plan for a DSL describing focus and allowing mode space to be computed statically rather than discovered dynamically. Finally, visibility/output of buffer contents remains a relatively unexplored issue, but clearly an important one.

Source code for our BA simulations is available from the authors.

7. REFERENCES
1. J. Campos and M. Harrison. Interaction engineering using the IVY tool. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, 2009.

2. Cardinal Health Inc. Alaris GP volumetric pump: directions for use. Technical report, Cardinal Health, 1180 Rolle, Switzerland, 2006.

3. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the 4th Annual Symposium on Logic in computer science*, 1989.

4. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

5. J. Coutaz. PAC, an object oriented model for dialog design. *INTERACT'87: Proceedings of the 2nd IFIP Conference on Human-Computer Interaction*, 1987.

6. A. Dix, J. Finlay, G. D. Abowd, and R. Beale. *Human-Computer Interaction*. Pearson, 3rd edition, 2004.

7. A. Gimblett and H. Thimbleby. User interface model discovery: towards a generic approach. In *EICS '10: Proceedings of the 2nd ACM SIGCHI symposium on Engineering interactive computing systems*, 2010.

8. D. Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3), 1987.

9. D. Harel and A. Naamad. The statemate semantics of statecharts. *ACM Trans. Softw. Eng. Methodol.*, 5, 1996.

10. F. Jahanian and A. Mok. Modechart: A specification language for real-time systems. *IEEE Transactions on Software Engineering*, 20, 1994.

11. J. Raskin. *The Humane Interface*. Addison Wesley, 2000.

12. B. Shneiderman and C. Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson, 5th edition, 2009.