COMPUTER ALGEBRA IN USER INTERFACE DESIGN ANALYSIS

Harold Thimbleby
UCLIC, UCL Interaction Centre
LONDON
http://www.uclic.ucl.ac.uk/harold

ABSTRACT

Computer algebra systems can do impressive mathematics that can help enormously in certain areas of formal HCI. This paper shows the ability to generate formal specifications, explore and generate theorems relevant to HCI needs, and how to do this automatically and reliably from existing, straight-forwardly programmed runnable systems. Conventional iterative design can modify these implementations, and we can then automatically redo formal analyses as the design is updated.

Keywords

Formal Methods in HCI, User Interface Design, Safety Critical Systems, Computer Algebra.

1. INTRODUCTION

Formal methods in HCI started in the late 1980s; the work to 1990 is summarised by Harrison and Thimbleby [4]. Formal theories of interaction were expressed (e.g., in the PIE model [5]), but there was little connection with actual implementation: the PIE model was theory, not for implementation. Most formal approaches did not scale well, and the discovery that simple properties, such as undo and modelessness, were inconsistent took further energy out of the pure formal approach. A good review of formal methods in HCI is Dix's 2003 chapter [1].

'Computer algebra' (CA) uses computers to do algebra. CAs now open up new possibilities in formal methods in HCI: they are fast and extremely competent, excelling human mathematicians in speed, accuracy, knowledge, and reproducibility [2]. The motivations of this paper include the following:

- To generate HCI-relevant formal specifications from ordinary programs. (This contribution appears to be unique; its relevance to formal HCI is exciting.)
- To enable user interfaces to be animated, debugged and iterated using standard and familiar programming techniques—to generate new specifications automatically at any time.
- To create abstract formal descriptions from full specifications, and use these to address focussed HCI questions.
- and to do all this using a standard, widely available, well documented notation.

This paper argues that using CA in HCI research and design is an effective approach for general analysis and prototyping. This paper uses *Mathematica* [11], a particular CA system with a full-featured programming language, GUI prototyping features, an outlining word processor including facilities to work with XML, and an enormous mathematical knowledgebase. Unlike many other approaches to formal methods in HCI, *Mathematica* is widely used, stable, well defined, and very well documented—with tutorials and reference books readily available. Our use of *Mathematica* is standard, and needs no special features. Although *Mathematica* has been used as a convenient programming language for simulating and analysing user interfaces [6, 7, 8], we are not aware that any CA system's algebraic facilities as such have been exploited in HCI. (Our approach can be directly contrasted with theorem proving and cognitive modelling, which unfortunately are both topics beyond the scope of a short paper.)

A claim that is hard to demonstrate in a static, textual paper is that the analysis can be redone in seconds with no additional effort if or when a system design is changed. In areas such as safety critical systems development—where one might be interested in the consequences of predictable classes of user error—formal analysis is essential. In the future, one can imagine design tools that provide the same power as a CA system, but without the

off-putting sophistication of a general purpose CA system. Separate work [3], which we do not describe here, is making the approach more accessible—cheaper (open source), and hiding the general mathematics behind a designer-oriented interface.

2. A FORMAL APPROACH

There are many ways a CA system can help formal HCI; for brevity we will discuss only one approach.

The state of a human-computer system can be described as a collection of parameters, where each action the user or computer does transforms some or all of those parameters. The user interacts to achieve their goals, and their goals are said to be achieved when certain parameters meet desired criteria. A user action such as "doing f" (e.g., moving a finger and pressing a button f) transforms the state space of the combined human-computer system; it can be represented by a mathematical function, $f \in \text{Action}: \text{State} \to \text{State}$. This very general approach to formalisation is routine [1].

The state space has structure, and functions may be specified by their transformations on one or more components of the state space. Turning now to working with a CA system, and specifically with *Mathematica*, a function f can be defined declaratively or imperatively. In a car we might formalise actions over a state space State $\subseteq$ Fuel $\times$ Speed $\times$ Position $\times$ Attention, and (for example) define a handbrake action in the form `f[{fuel_, speed_, User[leftHandPos_, rightHandPos_, attention_]}] := {fuel, 0, User[Handbrake, rightHandPos, Low]}`—here, to transform the speed component of a vehicle to zero when it is applied. Here, the user's left hand has moved to the handbrake, and the right hand position is unchanged.

In this very brief example, we have shown how a function can be defined over patterns of parameters, to return some function of those parameters, namely a new state. Such a specification can be run in a CA to simulate the device and its interaction.

Once a system is implemented our interest turns to HCI issues. Given a user task, what should a user do to achieve this task? This is a task/action mapping problem [12]. In our formalism, a task/action mapping is a function `how`: $P \times P' \to \text{Action}^*$ where P and P' are classes of states, and the action sequence gives states in P' for any states satisfying P. In safety critical systems, two important concerns are whether there is any task/action mapping that results in unsafe use, and whether there are 'simple' task/action mappings to achieve safe tasks.

What we need to do now is retrieve a formal specification from the implementation that can address these and other sorts of HCI question. Traditionally, this is considered infeasible: one normally obtains a program *from* a specification, not the other way around.

3. SIMULATION AND ANALYSIS

To proceed, we need a real example, simple enough for a short paper, but which raises non-trivial issues, and where results can be reproduced, checked and extended by the reader. Our case study is a Casio HS-8V handheld calculator; these are widely available, simple, standard handheld calculators which need no detailed discussion here—their user interface is well known, as is their task domain. Calculators also raise interesting HCI issues [9]. We have reverse engineered a HS-8V, and then implemented a simulation in the CA system. Note that the reverse engineering is not an issue with our approach: we did it because it gives us a real user interface based on a real product (i.e., our results can be checked by others), rather than a possibly partial interface based on a prototype or more trivial implementation which might not be widely available. Similar analyses might be made of other systems, robot control, navigation equipment, medical equipment, say. (The fact that calculators are mathematical in themselves is convenient for exposition but irrelevant to the approach.)

Our HS-8V device specification can be rendered by the CA as a fully working interactive prototype, just as if it had been programmed in a conventional programming environment—but because the simulation runs inside the CA system, we can do empirical usability work with the user interface design very easily.

The simulation generates a log of the functions a human or simulated user executes, which can be analysed or run again at any time, even concurrently during a user run—it could even be run in other simulations that start from different initial conditions (e.g., memory zero, memory non-zero). We get a composition of functions corresponding to user actions, for example, `CHSIGN` ∘ `ONE` ∘ `TWO` ∘ `DPOINT` ∘ `FIVE` ∘ `EIGHT` ∘ `MPLUS` ∘ `RT` ∘ `EIGHT`. Here, ∘ is the composition operator, and for example, `CHSIGN` is defined by `CHSIGN[{d_, m_, op_, arg_, mrc_, opc_}] := {-d, m, op, op[arg, d], 0, 0}` —which changes the sign of the display (`d`) and applies any outstanding operation (`op`, such as +) to the internal register and display.

We can obtain compositions with timings and other information if we wish. Of course, practically any system can run simulations and make logs. **The analyses we do next—from a fully working program—are normally impractical.**

As a composite function, the user log can be applied directly to *any* state to get the resultant

state. In a CA system, the state can be purely symbolic (representing any state whatsoever), partly symbolic (representing any state, but with assumptions), or a ground state (representing a particular state, such as the initial 'switch on' state). For example, we could ask: do the user's actions in this log *always* result in −85.21 being displayed? (Yes.) Do these actions *always* result in 2.82843 being in memory? (No.)

We represent any scenario as a function composing the scenario's actions with functions or symbolic values representing assumptions (either as initial conditions or distributed through it as invariants): thus we can easily check whether scenarios have intended outcomes and under what conditions. For example, the log (used above) composed with a function that fixes the initial memory to zero *always* results in 2.82843 in the memory, with no possible negative square root error whatever the initial display value.

How can a user store the currently displayed number in the memory? To answer this question, we start by defining this goal by `STORE[{display_, memory_, stuff__}] := {display, display, stuff};` Of course, users do not have access to `STORE` directly on the HS-8V; instead they have to solve a task/action mapping.

We define a set of available user actions, `funs = {ADD, EQ, MPLUS, CHSIGN, ...}` (which does not include `STORE`!), and call a function `how` (see web site) to compose functions to find a function equivalent to `STORE`. Assumptions can also be handled by `how`, to solve task/action mappings such as "if the user knows u, how can they v?" As a special case, we might set `funs` to be the union of the functions in a scenario and ask `how` to find an optimal solution with the same operations the user is assumed to know.

Users typically set the memory equal to zero prior to undertaking a task where they anticipate wanting to save any calculations to memory. The easiest way to model this behaviour is to define a task `zeroMemory` that specifies this. We then ask `how` what a user could do on this assumption. The solution for `STORE` is now found to be to just press M+. Separately, we can find the task/action mapping to do `zeroMemory`: press MRC twice, but this loses any number currently displayed. Thus a user cannot use the easy solution in the middle of a calculation.

There are some frequent user tasks where the user cannot avoid storing to a non-zero memory. Since the HS-8V is a non-algebraic calculator, a scenario involving solving sums like $(1{+}2){\times}(3{+}4){\times}(5{+}6)\ldots$ requires storing intermediate additions repeatedly: only the first partial sum can be done assuming a zero memory. Thus in general, the memory may not be zero, and the task is then *very* much harder for users—and impractical for most. The function `how` does a non-trivial symbolic search to solve the problem where almost all users would be stuck: all solutions for the `STORE` task on the HS-8V take at least four button presses (an optimal solution is to press − MRC M+ MRC), and there is *no* solution if the calculator is not in number ready mode.

Suppose (for concreteness) that M+ is pressed by accident. Can it be corrected by the user pressing M−? If so, the function for M− should be the inverse of M+. We try `funeq[MMINUS ∘ MPLUS, Identity]` to see if the composition of the functions corresponding to M+ and M− has the same effect as the identity function. But this does not evaluate to `True` for the HS-8V, as we might have hoped. It turns out that M− will only correct an accidental M+ *if* the M+ was pressed in the right mode.

4. LARGE SYSTEMS

A designer is rarely interested in the whole system all at once, and will mostly be wanting to make useful design decisions on components of the system. To illustrate, we define new notation ⟨`f`⟩ to find a linear operator over a specified vector space corresponding to the function `f`. Effectively ⟨`f`⟩ transforms a functional specification into matrices, which are much easier to analyse. Typically we will only be interested in components of the space that are visible to the user. In the case of the HS-8V calculator, the display and the memory are candidates—although the memory is invisible, the MRC key replaces the display with the memory contents, so users need to be aware of it.

We can transform the implementation of the calculator's M+ button to operate over this 2D real space $display \times memory$:

$$\begin{aligned} & (display,\ memory).\langle \texttt{MPLUS} \rangle \\ = \ & (display,\ memory).\begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \\ = \ & (display,\ display + memory) \end{aligned}$$

Note that the first CA step, $\langle \texttt{MPLUS} \rangle \Rightarrow \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, is an impressive simplification, transforming the complicated implementation of `MPLUS` (comparable to `CHSIGN` which was given above) to a simple matrix.

The composition of some functions have the same linear operator as the product of the operators taken individually (a homomorphism), so we can do much easier matrix multiplications to analyse the device—for example we evaluate ⟨`MMINUS ∘ MPLUS`⟩ == ⟨`MPLUS`⟩.⟨`MMINUS`⟩ and get `True`. This is a strong result, as the functional composition (on the left of `==`) is general in all device modes and states, whereas the matrix multiplication (on the right) is operating at the user awareness level.

As can be inferred from trying ⟨MRC⟩ ⇒ `No linear solution`, the MRC is modey, so it requires higher dimensions to represent as a linear operator. Put another way, if the linear display/memory model captures all that users model, then MRC has undesirable usability properties. As designers, we might want to simplify what MRC does. For example, by setting the 'has MRC button been hit' component to `False` we assert that MRC behaves as if pressed exactly once—and we can then find the 2×2 matrix for it. Call the restricted function `MRC1`. Now we can check the task/action mapping for the store task in a very different way. We project each user operation into matrices, then calculate the matrix product of the matrices taken in order:

$$\langle \texttt{MMINUS} \rangle . \langle \texttt{MRC1} \rangle . \langle \texttt{MPLUS} \rangle . \langle \texttt{MRC1} \rangle = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$$

The point is that given the matrices, this calculation becomes a trivial multiplication. Many such theorems can be found automatically. The result is a matrix, which when multiplying an initial state (*display*, *memory*) gives (*display*, *display*), as required (i.e., the memory has become equal to the initial display value).

5. Conclusions

It is clear that computer algebra is a valuable HCI research tool. What seem like simple user interface design issues have subtleties, which might be overlooked by conventional techniques.

In this paper we showed that a thorough user interface implementation can be manipulated to prove theorems about usability at various levels of detail; in particular, we showed how task/action mappings can be explored, and we showed that an arbitrary implementation can be projected down to a simple algebraic model, by way of example, to a linear algebra, which is known to be ideal for expressing and raising many user interface issues [10].

Points to emphasise are, first, that a CA system has generated insights (and can generate theorems and other properties) relevant to the user interface design, and, further, these formal statements can be derived reliably from an actual implementation. Secondly, it *works*. If we changed the implementation of the device, or changed the device to something different entirely, we could re-run the examples and get new results with no further ado.

The CA system, including the HS-8V program and all results shown in this paper, can be downloaded and modified in any way and the user interface analyses redone, developed and verified. See `http://www.uclic.ucl.ac.uk/harold/CA`

Acknowledgements Greg Abowd, Ann Blandford, Paul Cairns and Jeremy Gow made invaluable comments. Harold Thimbleby is a Royal Society-Wolfson Research Merit Award Holder.

References

[1] Dix, A. J. (2003). "Upside-Down ∀s and Algorithms—Computational Formalisms and Theory," *HCI Models, Theories and Frameworks*, 381–429, ed. Carroll, J. M., Morgan Kaufmann Pub.

[2] von zur Gathen, J. & Gerhard, J. (2003). *Modern Computer Algebra*, Cambridge University Press.

[3] Gow, J. & Thimbleby, H. (2004). "MAUI: An Interface Design Tool Based on Matrix Algebra," *ACM Conference on Computer Aided Design of User Interfaces*, CADUI **IV**, pp81–94.

[4] Harrison, M. D. & Thimbleby, H. (1990). *Formal Methods in Human Computer Interaction*, Cambridge University Press. (PB edition 1994.)

[5] Thimbleby, H. (1990). *User Interface Design*, Addison Wesley.

[6] Thimbleby, H. (1999). "Specification-led design for interface simulation, collecting use-data, interactive help, writing manuals, analysis, comparing alternative designs, etc," *Personal Technologies*, **4**(2):pp241–254.

[7] Thimbleby, H. (2000). "Analysis and simulation of user interfaces," *Human Computer Interaction 2000*, BCS Conference on Human-Computer Interaction, **XIV**, pp221–237.

[8] Thimbleby, H., Cairns, P. & Jones, M. (2001). "Usability analysis with Markov Models," *ACM Transactions on Computer Human Interaction*, **8**(2):pp99–132.

[9] Thimbleby, H. (2000). "Calculators are needlessly bad," *International Journal of Human-Computer Studies*, **52**(6):pp1031–1069.

[10] Thimbleby, H. (2004). "User interface design with matrix algebra," *ACM Transactions on Computer Human Interaction*, **11**(2):181–236.

[11] Wolfram, S. (1999). *The Mathematica Book*, 4ed. Cambridge University Press.

[12] Young, R. M. (1983). "Surrogates and Mappings: Two Kinds of Conceptual Models for Interactive Devices," Gentner, D. & Stevens, A. L. (eds.), *Mental models*, pp35–52, Hillsdale, NJ: Lawrence Erlbaum Assoc.