Interaction Programming: Next Steps

Harold Thimbleby
Future Interaction Technology Lab
Swansea University
Swansea, Wales, SA2 8PP
harold@thimbleby.net

Copyright is held by Harold Thimbleby, the author/owner.
CHI 2009, April 4 – 9, 2009, Boston, MA, USA
ACM 978-1-60558-247-4/09/04.

Abstract
Interaction programming bridges the gap between interaction design and programming, but it has not yet been related directly to mainstream development practice. This paper presents *UI model discovery tools* to enable existing systems and traditional development processes to benefit from interaction programming tools and methods.

Keywords
Discovery tools; interaction programming; model checking.

ACM Classification Keywords
H.5.2 User interfaces: Theory and methods.

Introduction

We have built some "user interface model discovery tools" that automatically construct rigorous models of interaction automatically, by effectively simulating the user in software. The approach works with existing interactive programs, and is of particular importance for safety critical interfaces; it is however not well-suited to "continuous" interaction (such as gestures or speech), although it can handle many of the UI features (such as menu choices) managing continuous interaction. This paper gives some examples for numeric entry (e.g., drug dose entry). Results presented in this paper were achieved automatically.

A conventional way to build quality software is to specify systems formally, prove specifications satisfy requirements, then rigorously derive working implementations. However, as a working system influences user behavior, requirements cannot be known until users have experience with one. Iterative design is the preferred way to build interactive systems, but changing a specification and re-deriving a program are too tedious and error-prone to do repeatedly, so systems *drift* from their specifications. Soon, many details of user interfaces become unknown, and systems delivered to users likely have latent defects that lie undetected indefinitely.

Drift can be addressed through source code analysis [1], where an abstract model is derived from the program. This gives many of the advantages of conventional formal software development, with the benefit of determining properties of the *actual* program being used. However, modern user interfaces are programmed in complex ways (e.g., using events, listeners, delegates, modes, etc) and source code analysis is impractical, ignores interaction details, or relies on assumptions (e.g., that Java Swing is used). Often much of the UI is implemented in libraries for which source code may not be available anyway.

Cognitive methods [cf 2], while addressing important complementary design criteria, have analogous problems to formal software engineering methods.

Press On: Principles of Interaction programming [5] is a recent book that summarizes problems with user interface design, provides a programming framework for analyzing interactive systems and avoiding interaction problems, and finally summarizes wider issues: going from programming to facing the real complexities of user requirements in the real world. In safety critical user interfaces (such as interactive medical devices) there is a great need for better interaction programming, as user problems with devices has been widely implicated in adverse clinical outcomes.

Although it provides a UI framework, *Press On* does not overcome the problems listed above. This paper presents a technique that links conventional development techniques with the advantages of interaction programming, as elaborated in *Press On*. The idea is to take the actual program and automatically discover the model that would have been used to define the system's interaction. We call such an approach a **UI model discovery tool**.

Since the manufacturer must anyway produce a working program, existing design processes and methods may be retained. One gains the advantages and insights of the *Press On* framework without any drawbacks. Indeed, the framework itself could be hidden from developers.

UI model discovery tools are related to previous work, particularly so-called *UI ripping*, which has been used for office applications [3] and Ajax web sites [4]. Ripping can quickly locate major defects (e.g., if the ripping fails!) but the idea in this paper is to discover models to analyze *detailed* UI structures from the actual programs that are being deployed.

The *Press On* framework

Production interactive systems have not been designed using the *Press On* framework: the framework is too

0.

In their initial state, most numeric entry systems display "0." leaving it unclear whether the user has or has not already pressed the decimal point. This creates an ambiguity. For example, if the user presses 5 next, the number entered will be either 5 or 0.5. For drug dose entry a factor of ten difference in the number entered for the *same* user action creates an unacceptable risk.

Note that the number displayed, "0." is arguably a syntax error and in any case would be forbidden by national regulations for drug dose entry.

Ideally the display would initially show 0; pressing 5 would change it to 5, and pressing . would change it to 0.X (say) as at this point the number entered ("0.") is in error until another digit is pressed.

recent, and systems continue being implemented with existing tools. There is no easy transition from current methods to better methods, because of the amount of legacy and investment in existing code. It is always going to be easier to make incremental changes to a system than to adopt a new approach, a potentially costly decision that involves refactoring, unless it is an approach that has large benefits and low adoption cost — such as model discovery tools.

Press On represents UIs as finite state machines, and uses the JavaScript Object Notation (JSON) to represent them. JSON is designed to represent objects in a well-defined and portable way. In *Press On*, JSON represents state machine transition matrices explicitly, and though this has the advantage of making analysis straightforward, it makes complex interactive systems impractical to manage by hand. Apart from explaining the techniques in the book, the real point of the framework is to analyze the JSON, not to see it, so its actual size or complexity is immaterial.

Discovering a UI specification

We have developed an API where the programmer declares the user operations (such as button press events) and invokes a discovery process by calling a function. The discovery process is dynamic: it runs the device to explore it. (A developer may choose to switch off real-world side-effects during the discovery process.) Currently the process creates a JSON or Mathematica specification. The discovery process uses a collection of callback functions and options to make the approach efficient and robust. We do not describe the programming details here, which also include model checking and methods to manage hidden mode problems (e.g., rechecking models against devices).

Obtaining a UI model from a program is theoretically hard, sometimes intractable, particularly if we do not require developers to make concessions in the way the program is written. However, if a discovery tool cannot reliably determine a UI model, then the system being developed *must* be very hard for a user to understand. Since the point is to make better UIs, using discovery tools can only help improve poor design!

Illustrative examples

We are interested in the safety properties of interactive medical devices, such as drug delivery systems. For example it is bad practice to enter drug doses in the form .5 gm or 5.0 gm, because the decimal point may be misread (the patient then gets doses of 5 or 50, in error by a factor of ten) yet few interactive devices enforce such rules [6]. Avoiding decimal point problems is a clear application of UI model discovery. It is not feasible to ensure correct decimal point handling by conventional user evaluation because there are too many cases to consider; every number — and every possible error — needs checking.

To make this paper more accessible to readers, rather than use drug delivery interfaces, we apply UI discovery to a full emulation of a widely-available calculator, the Casio HS-8V. As calculators are well known, the reader can check our results without understanding details of drug delivery. Calculators are also comparable in UI complexity to typical drug delivery systems. Indeed, from a computational perspective, calculators are *more* complex: they handle a wide range of numeric input (the HS-8V covers 10^{-7} to 10^{9} to 8 figures), far larger than necessary for drug dosage (at most 3 figures).

The diagrams in this paper, although accurate diagrams generated automatically, are for illustrative purposes only: they are reproduced at too small as scale to be easily readable!

In practice, visualization tools will be used to allow interactive zoom and scroll, and would allow much closer inspection.

With complex interactive systems — and the examples illustrated in this paper are on the verge of being too complex to understand from visualizations alone — the properties of the models will be checked and explored using tools.

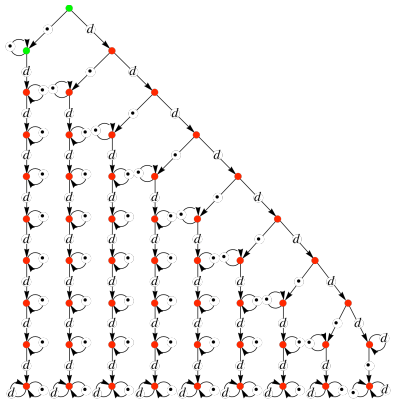

Figure 1. HS-8V numeric input device model, drawn as a transition diagram. Note that the model was obtained automatically directly from the interactive program.

Figure 1 shows the HS-8V's internal device model for handling numeric input. The specification for the graph was obtained automatically as described above. In the two states shown in green (top and top left) the calculator shows 0. on its display. In the top state the user has not pressed the decimal point, and there are two transitions from this state: if the user presses the decimal point (moving down and to the left), and if the user presses a digit (moving down and to the right). Note that this is an abstract model: e.g., *d* represents *any* digit, 0–9. To our knowledge, no UI models of this complexity have been extracted from fully-working programs before, and certainly not from programs where no concessions have been made to the model discovery process.

Figure 2 shows the HS-8V's user model for numeric input: device states are treated as equivalent if the user cannot distinguish them. Initially, the calculator displays 0., the state shown at the top of the diagram.

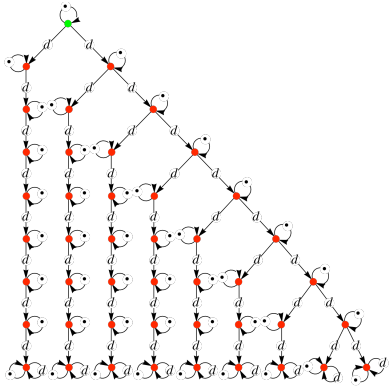

Figure 2. HS-8V numeric input user model, drawn as a transition diagram. The subtle but crucial differences with figure 1 can be reliably identified by using tools.

Pressing the decimal point does *not* change the state, however, but pressing a digit key does change the state in one of two possible ways. The user model is therefore non-deterministic: if a user presses a digit in the top state, the calculator may go left or right, *and the user will not know which*.

The two models differ in many details. Evidently the user's model is different from the device model, with the potential problem that users will misunderstand the treatment of numeric input. Probably no designer noticed this problem. Moreover, the defect that causes the non-determinism can be corrected *now the problem is identified*, then UI model discovery can easily check the defect is indeed corrected in every mode. Exactly the same design defect occurs on many drug delivery systems, where we also know that decimal point errors cause adverse events.

Infusion pumps, for example, have many modes where numbers can be entered, and model discovery can confirm that *in every mode* they behave as intended. Achieving this certainty with conventional usability methods is impractical: the two models here represent about 10^9 states (the *d* event represents any of ten digits), far beyond the time and resources available for conventional usability evaluation!

At this point in installing OSX, "Continue" is the *only* option the user can choose. The "Go back" button, although visible, is disabled.

A different example is illustrated by the Apple Mac system installation. The user inserts CDs to install the system, and part way through a very lengthy process the system says "click continue," but giving no other choices. In the calculator example, we had a single state with non-deterministic exits; here we have a single state with a single exit. Normally this is pointless, as the only choice the user has is to say *when* to continue [5]. In this case, the answer is always going to be *now* rather than waiting! (In contrast, for a fire extinguisher the user chooses to perform the only action *when* there is a fire.) This problematic type of state could be identified automatically and drawn to the attention of designers.

Possibly the pointless state is in the UI because, historically, several floppies were required for installation, and the original UI would have asked the user to be ready with disc 5, say. What is left now is legacy code that asks the user if they are ready when there is nothing to do! User testing would be impractical to find this problem: it takes hours to detect and might be missed (superficially the "Go back" option misleadingly appears to be available). Indeed, professional development *did* miss this problem. A discovery tool could search the entire UI state space, and *all* such latent problems could be addressed.

Potential implications

User interfaces are complex, and interaction problems are hard to find. Indeed, the prevalent culture is to ignore defects because one is always looking forward to the next upgrade or latest device that promises to fix all problems! In the safety critical domain, this attitude, whatever its full reasons, is unacceptable. "Small" and hard to find defects endanger life. Prior to the ideas discussed in this paper there was very little to help the designers and developers explore and discover such problems, and certainly no effective way to discover all UI design problems of a particular class. Even best practice in software engineering requires a huge manual effort to develop "correct" programs, and the once correct programs drift as they are iterated to improve their UI. Automatic model discovery methods are therefore essential design aids, and could help radically improve the quality of systems.

User testing requires humans, appropriate methods to record and analyze protocols, as well as appropriate ethical treatment of participants. Although any user testing can be insightful, thorough testing is slow and expensive. For many systems, exhaustive UI testing is infeasible because there is not enough time to try all necessary test cases: user testing rarely covers a design fully. Systematic testing with human users is also fraught with human error, not just with the system being tested, but with the whole process.

Automatic UI discovery raises no ethical issues, it does not demand human time for testing, and it can be used repeatedly with no extra cost as a design drifts. Once a developer has decided that certain issues or properties need analysis, tests can be repeated with ease on a range of device models or versions.

Ideally, conventional user evaluation would be combined with discovery tool analysis, and the two approaches would mutually inform each other. For example, user evaluation would identify types of problems, and model discovery would then be set up to ensure that those problems occur nowhere.

Recommendations for further work

Currently the discovery tool is a research method, but has shown its value in analyzing a range of systems and simulated systems. To refine it with a production system would be the next step.

In the future one imagines a discovery tool presenting a developer with a menu to select relevant properties to check, analyze, or audit. It may not be just that a designer is interested in a UI property, but requires it to lie in a specific range of values. Regulatory agencies may use discovery to test conformance to appropriate standards; manufacturers may use it in due diligence.

The main alternative to model discovery is to use source code analysis. (The arguments in favor of source code analysis also apply to discovery.) Key advantages of source code analysis are that it is fast, potentially more complete, and can obtain an abstract model directly [1]. Its problems were discussed above; source code analysis would only be reliable if an appropriate UI language was used, and this would impose a constraint the present approach avoids: the imposition of a particular programming regime on developers. But future work will bring these methods closer.

Conclusions

Interactive systems have transformed life, yet the details of programming them to interact reliably have not gained much attention: interaction is hard to visualize and hard to understand. Discovery tools allow designers and developers to retain their current processes and programming methodologies, but adds a powerful form of analysis that is essential in safety critical areas and beneficial in all other areas.

Acknowledgements

This work was supported by EPSRC grant EP/F020031. The author is a Royal Society-Leverhulme Trust Senior Research Fellow and acknowledges this generous support. Michael Harrison and Max Wilson made many formative comments on this paper.

Citations

[1] Holzman, G. J. & Smith, M. H. Software Model Checking: Extracting Verification Models from Source Code, *Proc. Software Testing, Verification and Reliability* (2001) **11**:65–79.

[2] John, B. E., Prevas, K., Salvucci, D. D. & Koedinger, K. Predictive Human Performance Modeling Made Easy, *Proc. ACM CHI* (2004) 455–462.

[3] Memon, A., Banerjee, I. & Nagarajan, A. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing, *Proc. Working Conf. on Reverse Engineering*, WCRE 2003 (2003) 260–269.

[4] Mesbah, A., Bozdag, E. & van Deursen, A. Crawling AJAX by Inferring User Interface State Changes, *Proc. 8th Int. Conf. Web Engineering*, ICWE08 (2008) 122–134.

[5] Thimbleby, H. *Press On: Principles of Interaction Programming*. MIT Press, Boston, Mass. USA (2007).

[6] Thimbleby, H. Ignorance of interaction programming is killing people, *ACM Interactions* (2008) 52–57, September+October.