Interaction Walkthrough: Evaluation of safety critical interactive systems

Harold Thimbleby

Department of Computer Science, Swansea University, Wales.
h.thimbleby@swansea.ac.uk

Abstract. Usability evaluation methods are a battery of techniques for assessing the usability of interactive systems or of proposed interactive systems. This paper describes a new evaluation method, particularly appropriate for evaluating safety critical and high quality user interfaces. The method can also be used for informing HCI research. The method is applied when a specification is available of an interactive system, or when a system (or prototype) is working.

1 Introduction

Human computer interaction is the science and practice of effective interactive systems, most often involving people and computers. Usability, more specifically, is the theory and application of methods to improve the processes and products of interactive system design. Humans are an object of study (for example, to advance the science of psychology), and interaction is an object of study (for example, using usability evaluation methods); interaction can also be arranged to present users with controlled, novel situations, and hence generate phenomena that demand psychological accounts. The conventional view, then, is that evaluation of interactive systems focuses on the user.

Less obvious is that the computer (or embedded computer), not just the human, is a useful object of study in its own right. Despite interactive systems being fully defined by program, their behaviour is often not fully understood, particularly when coupled with human use.

One might make an analogy from the well-known story of the Tacoma Narrows bridge. Human sciences are very interested in the experience, behaviour and performance of humans, for instance, drivers who use bridges — indeed, drivers on the Tacoma Narrows bridge reported motion sickness that could perhaps have stimulated new research. On the other hand, one might think that the engineering of a finished bridge is not an interesting object for study; after all, it has been built from specifications, so its design and properties are surely known in principle. Unfortunately, the Tacoma Narrows bridge experienced catastrophic failure in 1940 due to wind-induced vibration [15]. In other words, its engineering was *not* fully understood, and moreover, unknown engineering issues had a direct impact on its usability.

This paper proposes and demonstrates a new 'usability evaluation method' but uniquely focussing on the device engineering rather than on the user. The

benefits of this method are discussed, particularly for the safety critical domain. Analysis of the Tacoma Narrows bridge led to advances in dynamical systems, aerodynamics and so on; one may likewise hope that the methods proposed here for analysing interactive devices will lead in a similar way to advances in the theory and science of interactive devices.

1.1 Background

There are many usability evaluation methods, each appropriate to one or more stages of the design lifecycle. It is conventional to divide methods into three groups: test, inspection and inquiry. **Test methods** use representative users to work on typical tasks, and their performance is measured or otherwise assessed; testing requires a working system, or a prototype. **Inspection methods** instead use experts (typically HCI experts, but possibly software engineers or domain experts) to inspect a design; inspection can be done at any stage of design, from prototype to marketplace. **Inquiry methods** inquire into the users' preferences, desires, and behaviour, and try to establish the requirements of a design.

To this list should now be added **device methods**, such as this paper will describe below.

Conventional usability evaluation methods (UEMs) can be adjusted to need. For example think aloud can be used in at least three ways: introspection, normal think aloud, and cooperative evaluation. A UEM may be applied to the interactive system, the training material, the user manuals, or to the environment the user operates in. UEMs can be used in laboratories, in focus groups (perhaps with no system present), or in the wild. One can further run a UEM in anything from a purely informal way to an experimentally rigorous way — depending on whether one wants basic information to help a practical design, or whether one wants experimentally and statistically respectable information for, say, a scientific paper.

All UEMs are concerned primarily with the user experience, though they should be distinguished from user acceptance models [27], which are intended more to inform management or procurement questions than to improve design. On the other hand, *system* evaluation methods focus on the technical system design and are typically concerned with the reliability, integrity and safety of the system; such methods range from formal mathematical work (theorem proving, model checking) to informal procedures such as code walkthrough. There are many schools of software engineering (e.g., extreme programming, agile development), and many advocate different overall approaches. Nevertheless, software engineering methods assume the usability requirements are otherwise established, and evaluate conformance of the implemented system to its requirements.

UEMs are not without controversy. For example, Wixon [26] believes published papers on evaluations have failed the needs of usability professionals. Gray and Salzman [5, 6] compared many experiments with UEMs and found the standards of experimental work questionable, in some cases leading to erroneous claims.

All evaluation methods take stands on the questions of sampling, satisficing, frequency, severity and realism. There are other issues: such as obtaining quantitative or qualitative data. Generally, interactive systems and their use are complex; a UEM cannot hope to sample all of a system or all of its and its users' space of interactive behaviour (nor all users and all user personalities, etc). UEMs therefore sample a restricted space of the design — typically therefore taking some arbitrary or trying, on the other hand, to take some statistically valid sample of the space. This is of course difficult to do for interfaces intended to be culturally sensitive, enlarging the space again. Furthermore, the results of a UEM may be sensitive to the people doing the evaluation; some studies show little overlap in results when undertaken by different assessors [11] — interestingly, this paper ([11]) expresses surprise at assessor sensitivity, but it is more likely a consequence of the difficulty of sampling a large, unknown space uniformly.

A UEM may uncover problems in a design (e.g., there is no undo), but may not know how frequently this problem arises in use; that is, many UEMs cannot distinguish between risk and hazard. A UEM may identify a potential problem, but not be clear how severe a problem it is for users. For example, a system may have no undo, which is a potential problem, but users may have other work arounds — the user interface without undo may then be usefully simpler than one with extra features.

The more effort put into using a UEM, in principle the higher quality results that can be expected; different UEMs have different cost/benefit curves, and one may thus choose a break-off point, once encountering diminishing returns on effort. However, there is no guarantee that if a UEM identifies a problem that it can be fixed, or worse, that fixing it does not introduce other problems (which of course, at the time of fixing will be unknown and unevaluated problems).

Finally, there may be questions about realism. A device may be evaluated in a laboratory, but the conditions of actual use may be very different — this is a particularly tricky question with safety critical devices, which may be very hard and often unethical to evaluate under realistic conditions of use (e.g., where the user is in a state of stress).

1.2 Wider issues

Any review of UEMs, however brief, would not be complete without mentioning politics. Designs have many stakeholders, and UEMs generally attempt to promote the needs and values of users. These values may not be the values intended to be assessed or optimised by deploying the UEM. For example, in a safety critical environment, one may be more concerned with political fallout or public safety than the welfare of the actual operator, or one may be concerned with legal liability issues (hence, ironically, perhaps wanting an obfuscated design, so the user rather than the hardware seem to be at fault).

2 A UEM for safety critical systems

We now introduce a new UEM, primarily intended for safety critical systems. It can of course be used for any interactive system, but the sorts of safety insight obtained in terms of effort may be too costly for other domains — though the UEM can be used on parts of systems with less effort, and may thus provide insight into those parts of the system at reduced cost.

We call the method Interaction Walkthrough (IW) and it may most conveniently be contrasted with Cognitive Walkthrough (CW) and Program Walkthrough (PW)[1] — it 'fills the gap' between these two approaches.

Cognitive Walkthrough (CW) is a psychological inspection method that can be used at any stage using a prototype or real system; it does not require a fully functioning prototype; it does not require the involvement of users. CW is based on a certain psychological learning theory, and assumes the user sets a goal to be accomplished with the system. The user searches the interface for currently available action to achieve their goal and selects the action that seems likely to make progress toward the goal. It is assumed the user performs the selected action and evaluates the system's feedback for evidence that progress is being made toward the goal.

Assessors start with a general description of who the users will be and what relevant knowledge they possess, as well as a specific description of representative tasks to be performed and a list of the actions required to complete each of these tasks with the interface being evaluated.

The assessors then step through the tasks, taking notes in a structured way: evaluating at each step how difficult it is for the user to identify and operate the element relevant to their current goals, and evaluating how clearly the system provides feedback to that action. Each such step is classified a success or failure.

To classify each step, CW takes into consideration the user's (assumed) thought processes that contribute to decision making, such as memory load and ability to reason.

CW has been described well in the literature (e.g., [25]), and need not be reviewed further here, except to note that CW has many variants (e.g., [16]). The gist is that a psychologically informed evaluation is based on the user's behaviour.

In contrast, Program Walkthrough (PW) evaluates a design based on the computer's behaviour. Instead of psychological models or theories and user tasks and goals, the PW assessor has programming language theory and walks through program code to work out what the computer would do. In a sense, then, CW and PW are duals.

In PW the programmer (more often working in a team) works through program code, to convince themselves that the program does what it is supposed to do, and that all paths are executed under the right conditions. Often a program walkthrough will result in improved testing regimes, and of course better

[1] In the context of this paper, the alternative name Code Walkthrough might be confused with CW as Cognitive Walkthrough!

debugged code. Usually, PW is seen as a quality control process rather than an evaluation method *per se*; the assumption is that there is an 'ideal' program, that the current one should aspire to. Various techniques, including PW, are recruited to move the current implementation towards the ideal 'bug free' program.

Neither CW nor PW are concerned with the *interaction* itself, in the following sense. In CW, the concern is on the user, and in PW the concern is on the execution paths of the program — which may well have little to do with the interaction. That is, interaction is a side effect of running a program. The program does input and output, with lots of stuff in-between, but the interaction appears to the user as continuous (except when the computation is slow). If one looks at a fragment of code, one has to work out what the input and output relation is. In the worst case, there is a non-computable step from doing PW to knowing what the interaction is.

Consider the following trivial-looking Java method, which is not even interactive in any interesting sense:

```
void star(long n)
{ while( n > 1 ) n = n%2 == 0? n/2: 3*n+1;
  System.out.println("*");
}
```

If the user provides a value for `n`, does the method print a star? The code fragment runs the Collatz problem, and whether it always prints a star is an unsolved question (though it *will* always print a star if `n` is a Java `int`). This example makes a clear, if abstract, demonstration that a program walkthrough cannot in principle fully determine what the user will see the program doing. Indeed, real programs are far more complex: they are concurrent, event driven, and their behaviour is dependent on internal state, and so on.

Instead, we need to start outside the program ...

3 Interaction Walkthrough, IW

Many interactive devices are developed in a process roughly conforming to ISO13407: a prototype is built, tested and then refined, and then the final system is built (which may have further refinements). Typically, the prototype is developed in an informal way, in some convenient prototyping tool, and then the final system is developed in a programming language that is efficient on the target hardware. Sometimes documentation and other training material will be written, based on the same prototype, and developed concurrently with the target system development.

Interaction Walkthrough works analogously, but with different goals. From a working system, whether a prototype or target system, another system is developed. Thus, a *parallel system* is developed from the interaction behaviour of the system being evaluated.

Developing the parallel system requires the assessor to ask numerous questions, and to go over in detail many of the original design decisions — and

perhaps make some decisions explicit for the first time. To reprogram a system, the assessor needs to know whether features are instances of a common form, are different, or are identical ... and so on.

As the parallel system is built, the assessor will program in a new way, and therefore make independent decisions on the structure of the parallel program. The assessor makes notes of design questions and issues. Since the assessor is working independently of the original developers and programmers, new questions will be raised. For example: why are these two features *nearly* the same? Why weren't they programmed to be identical?

Inevitably, some (but by no means all) of the design issues raised by this phase of the IW will cover similar ground to a conventional heuristic evaluation [12] (e.g., noticing, if it is the case, that there is no undo).

The assessor will stop this phase of IW when either sufficient questions, or any 'killer' questions, have been raised, or when the rate of discovery has diminished and the rate of return is insufficient for the purposes of the evaluation. Generally, such discovery processes follow a Poisson distribution, and modelling the rate of discovery can inform a rational cut-off point [13].

Reprogramming a system or even a partial system, as required by IW, is not such a large undertaking as the original programming, since impossible and impractical features have already been eliminated. However reprogramming is still a considerable effort, and the assessor may choose to do a partial implementation, concentrating on certain features of interest. The assessor may use RAD tools, such as compiler-compilers, as well as *ad hoc* tools to both speed up the process and to ensure consistency.

Typically, it is not possible to determine exactly what the device's actual behaviour is merely by experimenting on it. The user manual and help material must also be used. Thus the assessor will discover discrepancies between the documentation and the target system. Moreover, the assessor will be using the user manual material with a keener eye than any user! Typically, user manuals have definitional problems that IW will expose.

At the end of the reimplementation phase of IW, the assessor has three products: a realistic simulation, a list of questions and/or critiques, and a deep understanding of the system (typically including a list of formal properties) — achieved more rapidly that other people on the design team. It is also worthwhile writing a *new* user manual for the simulation as well as documentation; writing manuals forces the assessor to work to a higher standard, and may also provide further insights into the design as a side-effect. It is also standard practice; if a safety critical device is being evaluated, the success of the evaluation should not depend on a particular member of staff.

Next the assessor involves users through cooperative evaluation [28]. Are the issues raised of significance to users in actual practice? Thus the list of questions and critiques is refined to a prioritised list.

The prioritised list can then be fed back into the iterative design process in the usual way. The IW implementation may also be iterated and made more realistic or more useful, for instance, for further cooperative evaluation.

3.1 Key steps of IW

In summary, IW is an UEM based on the following steps:

1. Clean reverse engineering, generally ignoring non-interaction issues, using a device, its user manual and training material.
2. Development of an accurate simulation.
3. Recording design questions and queries, arising from Steps 1 & 2.
4. Review with domain experts using the original device and/or simulation.

Steps 1–3 iterate until 'showstopper' or 'killer queries' are found, or the rate of discovery becomes insufficient. The whole process may be iterated; for example, working with end users may reveal short-comings in the simulation.

3.2 Variations and extensions

Once a simulation is available, it can be used for many purposes, such as training tests, collecting use data (which perhaps is too complex to collect from a real system), for random simulation, and so on. Other possibilities are to embed automatic UEMs [8] inside the system built for the IW exercise — many automated UEMs would interfere with the target system's operation, and so would be inappropriate to deploy without a different (in this case, IW) implementation. We do not consider these additional benefits strictly part of a UEM itself, but the 'added value' implies IW should be viewed as playing a wider part of the quality control and inquiry methods than merely finding certain types of usability problem.

An IW simulation can easily be built to generate a formal specification of the device in question, which can be used for model checking and other rigorous analyses. Indeed, if methods such as the Chinese Postman [2, 22] are used for reverse engineering, they in any case require an explicit graph of the device. Figure 2 shows an automatically drawn transition diagram, drawn from the specification generated by the IW program; the transition network has been checked for strong connectivity (as it happens, by one of the internal checks in the program itself, although the analysis could have been done by external programs). For example, the connectivity analysis in this example makes one node automatically highlighted as not being in the main component of the graph — thus showing either the device has a problem, or (as is the case here) the reverse engineering of it is not complete.

3.3 Relation of IW to software engineering

In typical software engineering processes, there is a progression (e.g., the spiral model or waterfall) from requirements to product. IW creates a parallel path, which does not start from the explicit specifications used in standard refinement.

Instead, the idea is effectively to 'rationally reconstruct' such specifications, insofar as they relate to user interaction, and to do so independently, without the implementation bias that would inevitably have crept into the original process.

Normal software engineering is concerned with validation and verification (V&V) of the product, with caveats that normally there are leaps of imagination connecting what can be formally established and what is actually implemented.For example, a model checking language like SMV does not lend itself to efficient implementation, so the target system is actually informally implemented, however rigorous its specification might have been. Instead, IW works from what has actually been implemented (not what should have been implemented), and works out its (interaction) specification, in fact, sufficient to reimplement some or all of its user interface. Thus it is likely that IW will bring rigour into interaction evaluation that, very likely, will have been omitted in the standard software development process.

4 Worked example

As an exercise in IW, a Graseby 3400 syringe pump and its user manual [4] was reverse engineered as a Java program. This worked example raises interesting design questions that were uncovered by IW — and on the whole, ones that were very unlikely to have been uncovered by other UEMs — and hence this example illustrates IW well. The Graseby 3400 is an extremely popular and established product (hence of some interest to evaluate by a new method) but this paper does not assess the risks, if any, of the issues mentioned.

This IW exercise developed three products:

- A photorealistic simulation; see Figure 1. The simulation only shows LEDs and the behaviour of the LCD panel; it does not show syringe activity.
- An automatically generated specification, used in particular to draw a transition diagram (which in fact is animated as the Java simulation is used); see Figure 2. This was developed to ensure the completeness of the simulation for the purposes of IW. In fact, not all of the 3400 was simulated, as it is a complex device.
- A list of 38 detailed design questions. The list of questions was generated as the reverse engineering proceeded. The reverse engineering phase was stopped when the rate of new question generation slowed and the process appeared unproductive. As will be seen, compared to a typical UEM, the questions are very detailed and technical, although presenting the full list is not the purpose of this paper.

Twelve anæsthetists and an NHS Pump Trainer (the person who trains pump operators) were interviewed, based on the list of 38 design questions generated during the reverse engineering phase. For reasons of space in this paper, we do not review all questions here; the paper is introducing IW, not reviewing any particular device except insofar as it helps illustrate IW.

Fig. 1. Screen shot of the Graseby simulation in bolus mode, with the 'start' LED flashing, indicating that the device is infusing. The image was made by holding the Graseby 3400 over a scanner.

The IW assessor then worked with a consultant anæsthetist in a detailed cooperative evaluation of the 3400, undertaken during a routine 3 hour operation.

Two incidents during the operation had been anticipated by some of the design questions. We now briefly summarise the design questions, then briefly summarise the use (in this case, clinical) perspective.

- The user manual states that the numeric keys are set up like a calculator. In fact, numbers have a slightly different syntax to an ordinary calculator, notably: (*i*) there is a *silent* timeout; (*ii*) decimal points zero the decimal fraction (whereas on a calculator, pressing the decimal point does not lose significant digits). See Table 1.
- Inaccurate entry of numeric data produces no error warnings (no beeps) and might (potentially) lead to adverse effects; see Table 1 for a summary.
- Numeric entry has a CANCEL button (which would be called AC or CE on a calculator), but there is no undo.
- The bolus feature has no explicit method of exit, but has a 10 second time-out. There are, however, 'spare' soft keys which might have been used for explicitly exiting bolus mode.
- Although there is a CANCEL button, there is no consistent escape from any mode.

The last case in Table 1 deserves further explanation. Conventional calculators do not ignore underflow unless a number already has more than (typically) 8 significant figures; in particular entering 0.009 would be handled correctly on all conventional calculators. Instead, the Graseby has a fixed number of decimal digits (1 or 2 depending on the mode), and always ignores underflow. As designed: underflow should be avoided by first choosing the correct units. For example, if the anæsthetist wants to enter 0.009mg/ml, they should instead (*as*

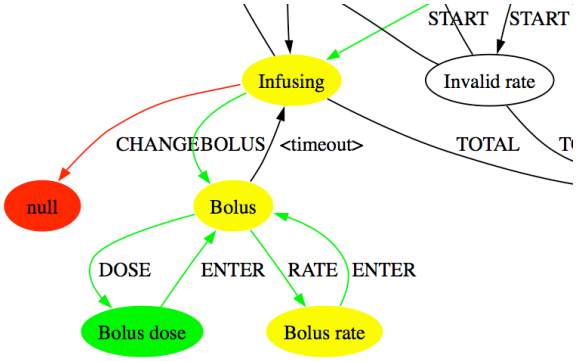

Fig. 2. Partial screen shot of the Graseby transition diagram, drawn by Dot. The diagram is updated after each user action, and it is coloured: the current state is green; transitions that have been tested are green; states that have been visited are yellow; undefined states and transitions are red — note that the simulation does not yet define the actions for changing the infusion rate.

the device is designed) enter 9μg/ml. This is an example of a design question, revealed by IW, which can then be presented to users, to see whether it is an actual (in this case, clinical) issue.

In a typical UEM these issues would easily be overlooked; in IW, they need examining closely otherwise a program cannot be written.

During the operation, the consultant anæsthetist entered the patient's weight incorrectly, as 8kg rather than 80kg.This was caused by a decimal point error.

The patient was infused continuously with Remifentanil, and, because of the length of the operation, the syringe needed to be changed. After the delay caused by a syringe change, the anæsthetist used the bolus feature, however he assumed switching the 3400 off and on again was the only way to get out of bolus mode and resume the infusion — switching off and on would of course have caused a further delay. He tried leaving the bolus mode by pressing various buttons, but of course this activity merely postponed the effect of the timeout which would have exited the bolus mode.

Neither of these incidents had any clinical significance (and they were not reported); in fact, they were quite routine issues. The anæsthetist noticed both, and had work-arounds. Nevertheless redesign of the 3400, along the lines anticipated in the IW, would have avoided both incidents. *Interestingly, such a redesign would not require any changes to any training material or the user manual; fixing the cases considered here would just make erroneous use simpler.*

In the larger context of improving the design, an open question is *how* the anæsthetist noticed his errors. For instance, an eye tracking experiment would help establish the relative effects of the LCD display's feedback to the tactile feedback of button pressing. Would improving the sound feedback help (in an already noisy operating theatre)? These are certainly worthwhile questions to

Key sequence	Effect	Comment
9999.42	999.42	*Numeric overflow loses digits* • Causes no error report (e.g., no beep) • Does not stop fraction entry • The number may seem correct to a quick glance
1.[pause]9	9.00	*Timeouts reset number entry* • No indication to user (e.g., no beep) • Number may be bigger than expected
111.[pause]9	9.00	*Timeouts reset number entry* • No indication to user (e.g., no beep) • Number may be smaller than expected
1.2.3	1.30	*Repeated decimals lose digits* • No indication to user (e.g., no beep)
0.009	0.00	*Numeric underflow is not rounded* • No indication to user (e.g., no beep)

Table 1. Partial list of potential issues to do with number entry. Column 2 shows the number displayed if a user presses the sequence of buttons indicated in Column 1 (provided there were no previous numeric buttons pressed in the current state).

investigate further, beyond the scope of IW, but questions raised by the process of IW.

5 Other examples

Several devices have previously been evaluated in methods approximating IW: microwave cooker [17]; video recorder [18]; fax [19]; mobile phone [20]; ticket machine [21]; calculator [23].

One might thus argue that IW (or an approximation thereof) gave or supported useful HCI design insights, of sufficient standard to contribute to these refereed publications.

There are *numerous* papers in the literature promoting new UEM or HCI methods using examples that are based on reverse engineered systems, that is, they are interpretable as 'hindsight-IW' case studies; however, these papers generally wish to make the claim that their methods should be used from specifications, not from reverse engineering, which may have been forced on them because their method was invented after the example product was build, or because the relevant form of specification was not available. The Palanque and Paternò edited book [14] is a case in point: it applies many formal HCI methods to existing browsers and browser features. For IW, the reverse engineering is integral to the process; it *adds* value, as explained above; whereas for most of these papers reverse engineering was used only incidentally, to ensure the proposed method could manage realistic cases (a detailed argument for this point of view is made in [24]).

6 Further work

There are several ways to develop the work proposed here.

The simulation was written in Java, but a more structured framework would be extremely beneficial (e.g., in VEG, Z, SMV etc), particularly for providing direct support for reverse engineering and output of specification texts for further analysis in standard tools; on the other hand, the Java included code for checking, generating Dot diagrams, and the Chinese Postman — in an ideal world, these would be features in interactive development frameworks.

To what extent are the insights of IW the investigator's insights, as opposed to insights derived from the IW process? The IW process should be tested and used with more analysts. Having a structured framework (as proposed above) would make IW more accessible to more people.

The recording of design queries should be integrated into the IW framework. Currently, the issues and questions were simply recorded in a textfile (for the worked example in this paper, we used LaTeX) and in comments in the Java program, but there was no tool-based management of them. In particular there was no explicit connection between the list of problems and the state space of the device — for example, had there been, diagrams like Figure 2 could be very helpfully annotated with actual and suspected problems.

7 Conclusions

As pointed out by Paul Cairns, IW is a programmer's version of Grounded Theory. Grounded Theory is, briefly, an inductive approach to build social theories grounded in systematically analysed empirical data [3]. In IW, the 'theory' is the resultant program (or, more precisely, the program is an implementation of the theory), and the empirical data is derived not from social data but from program behaviour.

There are six key reasons why IW reveals useful evaluation information that was unlikely to have been known without it:

1. The assessor is discovering a new specification by reverse engineering. This is more useful than re-examining the original specification (if there is one!) because the original specification and program inevitably contain traces of their development and the sequence of design decisions. For a user of the final system, the design rationale and its working out is irrelevant: the user is interested in the working product. Thus the IW assessor works from that product, and constructs a clean specification, without the 'historical taints' that CW would be enmeshed with.

2. Second, IW starts with a system that works. Many UEM inspection — CW amongst them — can work from prototypes, even non-functional prototypes such as story boards, which very likely will not have worked out details of how the real system is going to work, creating the problem Holmquist calls *cargo cult design* [7]. In safety critical areas it is crucial that the details get

evaluated and that nothing crucial is left to the imagination of the evaluators, who may be tempted to gloss details that have not been worked out for a prototype.

3. A third reason for the effectiveness of IW is that the assessor can choose any programming language or approach to construct the simulation — thus allowing them to work in a much more suitable programming environment than perhaps the implementors of the actual target system were able to. For example, the implementors of the target system may have had to use assembler or PIC code, whereas the IW assessor can use Java, SMV [9, 10], or Promela — or any special purpuse GUI languages, such as VEG [1] — as they wish. Using a 'nice' language with powerful abstraction features, particularly one with model checking, appropriate for the interactive simulation will highlight inconsistencies in the target design in a way that cannot be done on the target system.

4. Fourth, the simulation can embed arbitrary test, diagnostic and debugging information to help the assessor. For example, it can generate transition diagrams, diagnostic logs, and so forth: to help the assessor assess how much of the device has been simulated and tested, but also to give formal insight into the design, for instance to check properties. Algorithms such as the Chinese Postman [2, 22] can be used to ensure a complete coverage of the state space is achieved.

5. Fifth, the assessor can use a full range of modern model checking and theorem proving techniques to check the interface against desirable interaction properties. Loer and Harrison give persuasive examples [9, 10] of the effectiveness of this approach — indeed, their work is essentially IW, as they had to reverse engineer systems (though their approach is suited to early use in software production, on the assumption that formal specifications are available), and their systems do not have the performance to be production systems.

6. Finally, the *real* success of any UEM lies in its recommendations being adopted either in the product being evaluated or in some future product (or perhaps in persuasive argument in published papers — which will, in turn, affect future products). IW has the advantage over other UEMs that it is driven by reverse engineering software, and therefore is already half-way towards offering ideas that can contribute to changing the *software* of the product, such as how to refactor it. Other UEMs risk making suggestions that are superficially appealing (even based on empirical evidence) but which are too hard to implement and are therefore resisted by the very people who must make the intended changes.

Arguably, the greatest use of IW would be the software engineering insights it provides a system developer. Rather than rely on UEMs to identify usability problems that then need fixing, surely it is better to adopt a system development process that avoids as many problems as possible? To do so requires using suit-

able quality and quality assurance processes, which of course are often skipped under the pressures of industrial design. An IW process exploits best software engineering practice to build an accurate simulation of the system being evaluated; it is likely that the IW assessor is an expert programmer, and therefore brings to their evaluation practices that might be applied in normal design. If so, then IW can be an agent of change in improving work practice to avoid usability problems in *future* devices by encouraging the use of better processes. Indeed, in a typical industrial environment, identifying problems with today's product, which has already shipped, is of less interest than improving future products by learning from problems with today's product.

Further work that would be desirable is to compare the efficiency (significant problems identified per unit effort) of IW with other methods, as well as the organisational impact of IW (e.g., process improvements) against the impact of other methods.

Acknowledgements Harold Thimbleby is a Royal Society-Wolfson Research Merit Award Holder, and gratefully acknowledges this support. Paul Cairns (UCL) pointed out the useful analogy between IW and GT. Michael Harrison (Newcastle) and Matt Jones (Swansea) both provided many useful comments on the approach.

References

1. J. Berstel, S. Crepsi Reghizzi, G. Roussel & P. San Pietro, "A scalable formal method for design and automatic checking of user interfaces," *ACM Transactions on Software Engineerng and Methodology*, **14**(2):123-167, 2005.
2. W-H. Chen, "Test Sequence Generation from the Protocol Data Portion Based on the Selecting Chinese Postman Problem," *Information Processing Letters*, **65**(5):261–268, 1998.
3. B. G. Glaser & A. L. Strauss, *The Discovery of Grounded Theory: Strategies for Qualitative Research*, Chicago, 1967.
4. Graseby Medical Ltd., *Graseby 3400 Syringe Pump: Instruction Manual*, 2002.
5. W. D. Gray & M. C. Salzman, "Damaged merchandise? A review of experiments that compare usability evaluation methods," *Human-Computer Interaction*, **13**(3):203–261, 1998.
6. W. D. Gray & M. C. Salzman, "Repairing damaged merchandise: A rejoinder," *Human-Computer Interaction*, **13**(3):325–335, 1998.
7. Holmquist, L. E., "Prototyping: Generating ideas or cargo cult designs?" *ACM Interactions*, **12**(2):48–54, 2005.
8. M. Y. Ivory & M. A. Hearst, "The state of the art in automating usability evaluation of user interfaces," *ACM Computing Surveys*, **33**(4):470–516, 2001.
9. Loer, K. & Harrison, M., "Formal interactive systems analysis and usability inspection methods: Two incompatible worlds?, *Proceedings of the Interactive Systems. Design, Specification and Verification. 7th International Workshop*, DSV-IS 2000, Palanque, P. & Paternò, F. (eds) Lecture Notes in Computer Science, **1946**, 169–190, Springer-Verlag 2001.

10. Loer, K. & Harrison, M., "Towards usable and relevant model checking techniques for the analysis of dependable interactive systems," *Proceedings of the 17th IEEE International Conference on Automated Systems Engineering: ASE 2002*, Emmerich, W. & Wile, D. (eds), 223–226, 2002.
11. R. Molich, M. R. Ede, K. Kaasgaard & B. Karyukin, "Comparative usability evaluation", *Behaviour & Information Technology*, **23**(1):65–74, 2004.
12. J. Nielsen, *Usability engineering*, Academic Press, 1993.
13. J. Nielsen & T. K. Landauer, "A mathematical model of the finding of usability problems," *ACM SIGCHI conference on Human factors in computing systems*, 206–213, 1993.
14. Palanque, P. & Paternò, F., eds, *Formal Methods in Human Computer Interaction*, London, Springer-Verlag, 1997.
15. H. Petroski, *To Engineer is Human: The Role of Failure in Successful Design*, Vintage Books, 1992.
16. D. E. Rowley & D. G. Rhoades, "The Cognitive Jogthrough: A Fast-Paced User Interface Evaluation Procedure." *ACM CHI'92 Proceedings*, 389–395, 1992.
17. H. Thimbleby & I. H. Witten, "User Modelling as Machine Identification: New Design Methods for HCI," *Advances in Human Computer Interaction*, **IV**, D. Hix & H. R. Hartson, eds, 58–86, Ablex, 1993.
18. H. Thimbleby & with M. A. Addison, "Intelligent Adaptive Assistance and Its Automatic Generation," *Interacting with Computers*, **8**(1):51–68, 1996.
19. H. Thimbleby, "Specification-led Design for Interface Simulation, Collecting Use-data, Interactive Help, Writing Manuals, Analysis, Comparing Alternative Designs, etc," *Personal Technologies*, **4**(2):241–254, 1999.
20. H. Thimbleby, "Analysis and Simulation of User Interfaces," *Human Computer Interaction 2000*, BCS Conference on Human-Computer Interaction, S. McDonald, Y. Waern & G. Cockton, eds., **XIV**, 221–237, 2000.
21. H. Thimbleby, A. Blandford, P. Cairns, P. Curzon and M. Jones, "User Interface Design as Systems Design," *Proceedings People and Computers*, **XVI**, X. Faulkner, J. Finlay & F. Détienne, eds., 281–301, Springer, 2002.
22. H. Thimbleby, "The Directed Chinese Postman Problem," *Software — Practice & Experience*, **33**(11):1081–1096, 2003.
23. H. Thimbleby, "Computer Algebra in User Interface Design Analysis," *Proceedings BCS HCI Conference*, **2**, edited by A. Dearden and L. Watts, Research Press International, pp121–124, 2004.
24. H. Thimbleby, "User Interface Design with Matrix Algebra," *ACM Transactions on Computer-Human Interaction*, **11**(2):181–236, 2004.
25. C. Wharton, J. Rieman, C. Lewis & P. Polson, "The Cognitive Walkthrough Method: A Practictioner's Guide," in J. Nielsen & R. L. Mack, eds, *Usability Inspection Methods*, John Wiley and Sons, 1994.
26. D. R. Wixon, "Evaluating usability methods: why the current literature fails the practitioner," *Interactions*, **10**(4):28-34, 2003.
27. V. Venkatesh, M. G. Morris, G. B. Davis & F. D. Davis, "User acceptance of information technology: Toward a unified view," *MIS Quarterly*, **27**(3):425–478, 2003.
28. P. C. Wright & A. F. Monk, "The use of think-aloud evaluation methods in design," *ACM SIGCHI Bulletin*, **23**(1):55–57, 1991.